

# The Cook-Levin theorem

Frank J. Balbach

April 4, 2026

### Abstract

The Cook-Levin theorem states that deciding the satisfiability of Boolean formulas in conjunctive normal form is  $\mathcal{NP}$ -complete. This entry formalizes a proof of this theorem based on the textbook *Computational Complexity: A Modern Approach* by Arora and Barak. It contains definitions of deterministic multi-tape Turing machines, the complexity classes  $\mathcal{P}$  and  $\mathcal{NP}$ , polynomial-time many-one reduction, and the decision problem **SAT**. For the  $\mathcal{NP}$ -hardness of **SAT**, the proof first shows that every polynomial-time computation can be performed by a two-tape oblivious Turing machine. An  $\mathcal{NP}$  problem can then be reduced to **SAT** by a polynomial-time Turing machine that encodes computations of the problem's oblivious two-tape verifier Turing machine as formulas in conjunctive normal form.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Outline . . . . .	4
1.2	Related work . . . . .	5
1.3	The core concepts . . . . .	5
<b>2</b>	<b>Turing machines</b>	<b>6</b>
2.1	Basic definitions . . . . .	6
2.1.1	Multi-tape Turing machines . . . . .	6
2.1.2	Computing a function . . . . .	17
2.1.3	Pairing strings . . . . .	19
2.1.4	Big-Oh and polynomials . . . . .	20
2.2	Increasing the alphabet or the number of tapes . . . . .	22
2.2.1	Enlarging the alphabet . . . . .	22
2.2.2	Increasing the number of tapes . . . . .	23
2.3	Combining Turing machines . . . . .	24
2.3.1	Relocated machines . . . . .	24
2.3.2	Sequences . . . . .	25
2.3.3	Branches . . . . .	27
2.3.4	Loops . . . . .	28
2.3.5	A proof method . . . . .	30
2.4	Elementary Turing machines . . . . .	30
2.4.1	Clean tapes . . . . .	31
2.4.2	Moving tape heads . . . . .	32
2.4.3	Copying and translating tape contents . . . . .	34
2.4.4	Writing single symbols . . . . .	41
2.4.5	Writing a symbol multiple times . . . . .	43
2.4.6	Moving to the start of the tape . . . . .	44
2.4.7	Erasing a tape . . . . .	46
2.4.8	Writing a symbol sequence . . . . .	47
2.4.9	Setting the tape contents to a symbol sequence . . . . .	47
2.4.10	Comparing two tapes . . . . .	48
2.4.11	Computing the identity function . . . . .	51
2.5	Memorizing in states . . . . .	52
2.6	Composing functions . . . . .	57
2.7	Arithmetic . . . . .	62
2.7.1	Binary numbers . . . . .	63
2.7.2	Incrementing . . . . .	69
2.7.3	Decrementing . . . . .	71
2.7.4	Addition . . . . .	74
2.7.5	Multiplication . . . . .	80
2.7.6	Powers . . . . .	86
2.7.7	Monomials . . . . .	88
2.7.8	Polynomials . . . . .	90
2.7.9	Division by two . . . . .	93
2.7.10	Modulo two . . . . .	98
2.7.11	Boolean operations . . . . .	98
2.8	Lists of numbers . . . . .	100

2.8.1	Representation as symbol sequence . . . . .	100
2.8.2	Moving to the next element . . . . .	103
2.8.3	Appending an element . . . . .	105
2.8.4	Computing the length . . . . .	106
2.8.5	Extracting the $n$ -th element . . . . .	108
2.8.6	Finding the previous position of an element . . . . .	111
2.8.7	Checking containment in a list . . . . .	116
2.8.8	Creating lists of consecutive numbers . . . . .	119
2.8.9	Creating singleton lists . . . . .	122
2.8.10	Extending with a list . . . . .	123
2.9	Lists of lists of numbers . . . . .	125
2.9.1	Representation as symbol sequence . . . . .	125
2.9.2	Appending an element . . . . .	127
2.9.3	Extending with a list . . . . .	128
2.9.4	Moving to the next element . . . . .	130
2.10	Mapping between a binary and a quaternary alphabet . . . . .	131
2.10.1	Encoding and decoding . . . . .	131
2.10.2	Turing machines for encoding and decoding . . . . .	133
2.11	Symbol sequence operations . . . . .	139
2.11.1	Checking for being over an alphabet . . . . .	139
2.11.2	The length of the input . . . . .	141
2.11.3	Whether the length is even . . . . .	143
2.11.4	Checking for ends-with or empty . . . . .	145
2.11.5	Stripping trailing symbols . . . . .	146
2.11.6	Writing arbitrary length sequences of the same symbol . . . . .	148
2.11.7	Extracting the elements of a pair . . . . .	150
2.12	Well-formedness of lists . . . . .	155
2.12.1	A criterion for well-formed lists . . . . .	155
2.12.2	A criterion for well-formed lists of lists . . . . .	157
2.12.3	A Turing machine to check for subsequences of length two . . . . .	157
2.12.4	Checking well-formedness for lists . . . . .	161
2.12.5	Checking well-formedness for lists of lists . . . . .	164
<b>3</b>	<b>Time complexity</b> . . . . .	<b>168</b>
3.1	The time complexity classes DTIME, $\mathcal{P}$ , and $\mathcal{NP}$ . . . . .	168
3.2	Restricting verifiers to one-bit output . . . . .	169
3.3	$\mathcal{P}$ is a subset of $\mathcal{NP}$ . . . . .	171
3.4	More about $\mathcal{P}$ , $\mathcal{NP}$ , and reducibility . . . . .	172
<b>4</b>	<b>Satisfiability</b> . . . . .	<b>173</b>
4.1	The language SAT . . . . .	173
4.1.1	CNF formulas and satisfiability . . . . .	173
4.1.2	Predicates on assignments . . . . .	175
4.1.3	Representing CNF formulas as strings . . . . .	176
4.2	SAT is in $\mathcal{NP}$ . . . . .	178
4.2.1	Verifying SAT instances . . . . .	178
4.2.2	A Turing machine for verifying formulas . . . . .	180
4.2.3	A Turing machine for verifying SAT instances . . . . .	188
<b>5</b>	<b>Obliviousness</b> . . . . .	<b>196</b>
5.1	Oblivious Turing machines . . . . .	196
5.1.1	Traces and head positions . . . . .	197
5.1.2	Increasing the number of tapes . . . . .	198
5.1.3	Combining Turing machines . . . . .	198
5.1.4	Traces for elementary Turing machines . . . . .	199
5.1.5	Memorizing in states . . . . .	205
5.2	Constructing polynomials in polynomial time . . . . .	206
5.2.1	Initializing the output tape . . . . .	207
5.2.2	Multiplying by the input length . . . . .	209

5.2.3	Appending a fixed number of symbols	216
5.2.4	Polynomials of higher degree	217
5.3	Existence of two-tape oblivious Turing machines	219
5.3.1	Encoding multiple tapes into one	221
5.3.2	Construction of the simulator Turing machine	223
5.3.3	Semantics of the Turing machine	233
5.3.4	Shrinking the Turing machine to two tapes	257
5.3.5	Time complexity	258
5.3.6	Obliviousness	258
5.4	$\mathcal{NP}$ and obliviousness	258
<b>6</b>	<b>Reducing <math>\mathcal{NP}</math> languages to SAT</b>	<b>260</b>
6.1	Introduction	260
6.1.1	Preliminaries	260
6.1.2	Construction of the CNF formula	261
6.2	Auxiliary CNF formulas	265
6.3	The functions <i>inputpos</i> and <i>prev</i>	266
6.4	Snapshots	270
6.5	The CNF formula $\Phi$	274
6.6	Correctness of the formula	275
6.6.1	$\Phi$ satisfiable implies $x \in L$	275
6.6.2	$x \in L$ implies $\Phi$ satisfiable	275
<b>7</b>	<b>Auxiliary Turing machines for reducing <math>\mathcal{NP}</math> languages to SAT</b>	<b>278</b>
7.1	Generating literals	278
7.2	A Turing machine for relabeling formulas	279
7.2.1	The length of relabeled formulas	279
7.2.2	Relabeling clauses	280
7.2.3	Relabeling CNF formulas	284
7.3	Listing the head positions of a Turing machine	287
7.3.1	Simulating and logging head movements	287
7.3.2	Adjusting head position counters	288
7.3.3	Listing the head positions	290
7.4	A Turing machine for $\Psi$ formulas	295
7.4.1	The general case	296
7.4.2	For intervals	300
7.5	A Turing machine for $\Upsilon$ formulas	303
7.5.1	A Turing machine for singleton clauses	303
7.5.2	A Turing machine for $\Upsilon(\gamma_i)$ formulas	306
7.6	Turing machines for the parts of $\Phi$	309
7.6.1	A Turing machine for $\Phi_0$	310
7.6.2	A Turing machine for $\Phi_1$	313
7.6.3	A Turing machine for $\Phi_2$	315
7.6.4	Turing machines for $\Phi_3, \Phi_4,$ and $\Phi_5$	318
7.6.5	A Turing machine for $\Phi_6$	321
7.6.6	A Turing machine for $\Phi_7$	325
7.6.7	A Turing machine for $\Phi_8$	327
7.6.8	A Turing machine for $\Phi_9$	329
<b>8</b>	<b>Turing machines for reducing <math>\mathcal{NP}</math> languages to SAT</b>	<b>345</b>
8.1	Turing machines for parts of $\Phi$ revisited	345
8.2	A Turing machine for initialization	349
8.3	The actual Turing machine computing the reduction	357
8.4	SAT is $\mathcal{NP}$ -complete	386

# Chapter 1

## Introduction

The Cook-Levin theorem states that the problem **SAT** of deciding the satisfiability of Boolean formulas in conjunctive normal form is  $\mathcal{NP}$ -complete [4, 10]. This article formalizes a proof of this theorem based on the textbook *Computational Complexity: A Modern Approach* by Arora and Barak [2].

### 1.1 Outline

We start out in Chapter 2 with a definition of multi-tape Turing machines (TMs) slightly modified from Arora and Barak’s definition. The remainder of the chapter is devoted to constructing ever more complex machines for arithmetic on binary numbers, evaluating polynomials, and performing basic operations on lists of numbers and even lists of lists of numbers.

Specifying Turing machines and proving their correctness and running time is laborious at the best of times. We slightly alleviate the seemingly inevitable tedium of this by defining elementary reusable Turing machines and introducing ways of composing them sequentially as well as in if-then-else branches and while loops. Together with the representation of natural numbers and lists, we thus get something faintly resembling a structured programming language of sorts.

In Chapter 3 we introduce some basic concepts of complexity theory, such as  $\mathcal{P}$ ,  $\mathcal{NP}$ , and polynomial-time many-one reduction. Following Arora and Barak the complexity class  $\mathcal{NP}$  is defined via verifier Turing machines rather than nondeterministic machines, and so the deterministic TMs introduced in the previous chapter suffice for all definitions. To flesh out the chapter a little we formalize obvious proofs of  $\mathcal{P} \subseteq \mathcal{NP}$  and the transitivity of the reducibility relation, although neither result is needed for proving the Cook-Levin theorem.

Chapter 4 introduces the problem **SAT** as a language over bit strings. Boolean formulas in conjunctive normal form (CNF) are represented as lists of clauses, each consisting of a list of literals encoded in binary numbers. The list of lists of numbers “data type” defined in Chapter 2 will come in handy at this point.

The proof of the Cook-Levin theorem has two parts: Showing that **SAT** is in  $\mathcal{NP}$  and showing that **SAT** is  $\mathcal{NP}$ -hard, that is, that every language in  $\mathcal{NP}$  can be reduced to **SAT** in polynomial time. The first part, also proved in Chapter 4, is fairly easy: For a satisfiable CNF formula, a satisfying assignment can be given in roughly the size of the formula, because only the variables in the formula need be assigned a truth value. Moreover whether an assignment satisfies a CNF formula can be verified easily.

The hard part is showing the  $\mathcal{NP}$ -hardness of **SAT**. The first step (Chapter 5) is to show that every polynomial-time computation on a multi-tape TM can be performed in polynomial time on a two-tape *oblivious* TM. Oblivious means that the sequence of positions of the Turing machine’s tape heads depends only on the *length* of the input. Thus any language in  $\mathcal{NP}$  has a polynomial-time two-tape oblivious verifier TM. In Chapter 6 the proof goes on to show how the computations of such a machine can be mapped to CNF formulas such that a CNF formula is satisfiable if and only if the underlying computation was for a string in the language **SAT** paired with a certificate. Finally in Chapter 7 and Chapter 8 we construct a Turing machine that carries out the reduction in polynomial time.

## 1.2 Related work

The Cook-Levin theorem has been formalized before. Gamboa and Cowles [8] present a formalization in ACL2 [3]. They formalize  $\mathcal{NP}$  and reducibility in terms of Turing machines, but analyze the running time of the reduction from  $\mathcal{NP}$ -languages to SAT in a different, somewhat ad-hoc, model of computation that they call “the major weakness” of their formalization.

Employing Coq [13], Gähler and Kunze [7] define  $\mathcal{NP}$  and reducibility in the computational model “call-by-value  $\lambda$ -calculus L” introduced by Forster and Smolka [6]. They show the  $\mathcal{NP}$ -completeness of SAT in this framework. Turing machines appear in an intermediate problem in the chain of reductions from  $\mathcal{NP}$  languages to SAT, but are not used to show the polynomiality of the reduction. Nevertheless, this is likely the first formalization of the Cook-Levin theorem where both the complexity theoretic concepts and the proof of the polynomiality of the reduction use the same model of computation.

With regards to Isabelle, Xu et al. [15] provide a formalization of single-tape Turing machines with a fixed binary alphabet in the computability theory setting and construct a universal TM. While I was putting the finishing touches on this article, Dalvit and Thiemann [5] published a formalization of (deterministic and nondeterministic) multi-tape and single-tape Turing machines and showed how to simulate the former on the latter with quadratic slowdown. Moreover, Thiemann and Schmidinger [14] prove the  $\mathcal{NP}$ -completeness of the Multiset Ordering problem, without, however, proving the polynomial-time computability of the reduction.

This article uses Turing machines as model of computation for both the complexity theoretic concepts and the running time analysis of the reduction. It is thus most similar to Gähler and Kunze’s work, but has a more elementary, if not brute-force, flavor to it.

## 1.3 The core concepts

The proof of the Cook-Levin theorem awaits us in Section 8.4 on the very last page of this article. The way there is filled with definitions of Turing machines, correctness proofs for Turing machines, and running time-bound proofs for Turing machines, all of which can easily drown out the more relevant concepts. For instance, for verifying that the theorem on the last page really is the Cook-Levin theorem, only a small fraction of this article is relevant, namely the definitions of  $\mathcal{NP}$ -completeness and of SAT. Recursively breaking down these definitions yields:

- $\mathcal{NP}$ -completeness: Section 3.1
  - languages: Section 3.1
  - $\mathcal{NP}$ -hard: Section 3.1
    - \*  $\mathcal{NP}$ : Section 3.1
      - Turing machines: Section 2.1.1
      - computing a function: Section 2.1.2
      - pairing strings: Section 2.1.3
      - Big-Oh, polynomial: Section 2.1.4
    - \* polynomial-time many-one reduction: Section 3.1
- SAT: Section 4.1.3
  - literal, clause, CNF formula, assignment, satisfiability: Section 4.1.1
  - representing CNF formulas as strings: Section 4.1.3
    - \* string: Section 2.1.1
    - \* CNF formula: Section 4.1.1
    - \* mapping between symbols and strings: Section 2.1.2
    - \* mapping between binary and quaternary alphabets: Section 2.10.1
    - \* lists of lists of natural numbers: Section 2.9.1
      - binary representation of natural numbers: Section 2.7.1
      - lists of natural numbers: Section 2.8.1

In other words the Sections 2.1, 2.7.1, 2.8.1, 2.9.1, 2.10.1, 3.1, 4.1.1, and 4.1.3 cover all definitions for formalizing the statement “SAT is  $\mathcal{NP}$ -complete”.

# Chapter 2

## Turing machines

This chapter introduces Turing machines as a model of computing functions within a running-time bound. Despite being quite intuitive, Turing machines are notoriously tedious to work with. And so most of the rest of the chapter is devoted to making this a little easier by providing means of combining TMs and a library of reusable TMs for common tasks.

The basic idea (Sections 2.1 and 2.2) is to treat Turing machines as a kind of GOTO programming language. A state of a TM corresponds to a line of code executing a rather complex command that, depending on the symbols read, can write symbols, move tape heads, and jump to another state (that is, line of code). States are identified by line numbers. This makes it easy to execute TMs in sequence by concatenating two TM “programs”. On top of the GOTO implicit in all commands, we then define IF and WHILE in the traditional way (Section 2.3). This makes TMs more composable.

The interpretation of states as line numbers deprives TMs of the ability to memorize values “in states”, for example, the carry bit during a binary addition. In Section 2.5 we recover some of this flexibility.

Being able to combine TMs is helpful, but we also need TMs to combine. This takes up most of the remainder of the chapter. We start with simple operations, such as moving a tape head to the next blank symbol or copying symbols between tapes (Section 2.4). Extending our programming language analogy for more complex TMs, we identify tapes with variables, so that a tape contains a value of a specific type, such as a number or a list of numbers. In the remaining Sections 2.7 to 2.12 we define these “data types” and devise TMs for operations over them.

It would be an exaggeration to say all this makes working with Turing machines easy or fun. But at least it makes TMs somewhat more feasible to use for complexity theory, as witnessed by the subsequent chapters.

### 2.1 Basic definitions

```
theory Basics
imports Main
begin
```

While Turing machines are fairly simple, there are still a few parts to define, especially if one allows multiple tapes and an arbitrary alphabet: states, tapes (read-only or read-write), cells, tape heads, head movements, symbols, and configurations. Beyond these are more semantic aspects like executing one or many steps of a Turing machine, its running time, and what it means for a TM to “compute a function”. Our approach at formalizing all this must look rather crude compared to Dalvit and Thiemann’s [5], but still it does get the job done.

For lack of a better place, this section also introduces a minimal version of Big-Oh, polynomials, and a pairing function for strings.

#### 2.1.1 Multi-tape Turing machines

Arora and Barak [2, p. 11] define multi-tape Turing machines with these features:

- There are  $k \geq 2$  infinite one-directional tapes, and each has one head.

- The first tape is the input tape and read-only; the other  $k - 1$  tapes can be written to.
- The tape alphabet is a finite set  $\Gamma$  containing at least the blank symbol  $\square$ , the start symbol  $\triangleright$ , and the symbols  $\mathbf{0}$  and  $\mathbf{1}$ .
- There is a finite set  $Q$  of states with start state and halting state  $q_{start}, q_{halt} \in Q$ .
- The behavior is described by a transition function  $\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^{k-1} \times \{L, S, R\}^k$ . If the TM is in a state  $q$  and the symbols  $g_1, \dots, g_k$  are under the  $k$  tape heads and  $\delta(q, (g_1, \dots, g_k)) = (q', (g'_1, \dots, g'_k), (d_1, \dots, d_k))$ , then the TM writes  $g'_1, \dots, g'_k$  to the writable tapes, moves the tape heads in the direction (Left, Stay, or Right) indicated by the  $d_1, \dots, d_k$  and switches to state  $q'$ .

## Syntax

An obvious data type for the direction a tape head can move:

**datatype** *direction* = *Left* | *Stay* | *Right*

We simplify the definition a bit in that we identify both symbols and states with natural numbers:

- We set  $\Gamma = \{0, 1, \dots, G - 1\}$  for some  $G \geq 4$  and represent the symbols  $\square$ ,  $\triangleright$ ,  $\mathbf{0}$ , and  $\mathbf{1}$  by the numbers 0, 1, 2, and 3, respectively. We represent an alphabet  $\Gamma$  by its size  $G$ .
- We let the set of states be of the form  $\{0, 1, \dots, Q\}$  for some  $Q \in \mathbb{N}$  and set the start state  $q_{start} = 0$  and halting state  $q_{halt} = Q$ .

The last item presents a fundamental difference to the textbook definition, because it requires that Turing machines with  $q_{start} = q_{halt}$  have exactly one state, whereas the textbook definition allows them arbitrarily many states. However, if  $q_{start} = q_{halt}$  then the TM starts in the halting state and thus does not actually do anything. But then it does not matter if there are other states besides that one start/halting state. Our simplified definition therefore does not restrict the expressive power of TMs. It does, however, simplify composing them.

The type *nat* is used for symbols and for states.

**type-synonym** *state* = *nat*

**type-synonym** *symbol* = *nat*

It is confusing to have the numbers 2 and 3 represent the symbols  $\mathbf{0}$  and  $\mathbf{1}$ . The next abbreviations try to hide this somewhat. The glyphs for symbols number 4 and 5 are chosen arbitrarily. While we will encounter Turing machines with huge alphabets, only the following symbols will be used literally:

**abbreviation** (*input*) *blank-symbol* :: *nat* ( $\langle \square \rangle$ ) **where**  $\square \equiv 0$

**abbreviation** (*input*) *start-symbol* :: *nat* ( $\langle \triangleright \rangle$ ) **where**  $\triangleright \equiv 1$

**abbreviation** (*input*) *zero-symbol* :: *nat* ( $\langle \mathbf{0} \rangle$ ) **where**  $\mathbf{0} \equiv 2$

**abbreviation** (*input*) *one-symbol* :: *nat* ( $\langle \mathbf{1} \rangle$ ) **where**  $\mathbf{1} \equiv 3$

**abbreviation** (*input*) *bar-symbol* :: *nat* ( $\langle | \rangle$ ) **where**  $| \equiv 4$

**abbreviation** (*input*) *sharp-symbol* :: *nat* ( $\langle \# \rangle$ ) **where**  $\# \equiv 5$

**unbundle** *no abs-syntax*

Tapes are infinite in one direction, so each cell can be addressed by a natural number. Likewise the position of a tape head is a natural number. The contents of a tape are represented by a mapping from cell numbers to symbols. A *tape* is a pair of tape contents and head position:

**type-synonym** *tape* = (*nat*  $\Rightarrow$  *symbol*)  $\times$  *nat*

Our formalization of Turing machines begins with a data type representing a more general concept, which we call *machine*, and later adds a predicate to define which machines are *Turing* machines. In this generalization the number  $k$  of tapes is arbitrary, although machines with zero tapes are of little interest. Also, all tapes are writable and the alphabet is not limited, that is,  $\Gamma = \mathbb{N}$ . The transition function becomes  $\delta: \{0, \dots, Q\} \times \mathbb{N}^k \rightarrow \{0, \dots, Q\} \times \mathbb{N}^k \times \{L, S, R\}^k$  or, saving us one occurrence of  $k$ ,  $\delta: \{0, \dots, Q\} \times \mathbb{N}^k \rightarrow \{0, \dots, Q\} \times (\mathbb{N} \times \{L, S, R\})^k$ .

The transition function  $\delta$  has a fixed behavior in the state  $q_{halt} = Q$  (namely making the machine do nothing). Hence  $\delta$  needs to be specified only for the  $Q$  states  $0, \dots, Q-1$  and thus can be given as a sequence  $\delta_0, \dots, \delta_{Q-1}$  where each  $\delta_q$  is a function

$$\delta_q: \mathbb{N}^k \rightarrow \{0, \dots, Q\} \times (\mathbb{N} \times \{L, S, R\})^k. \quad (2.1)$$

Going one step further we allow the machine to jump to any state in  $\mathbb{N}$ , and we will treat any state  $q \geq Q$  as a halting state. The  $\delta_q$  are then

$$\delta_q: \mathbb{N}^k \rightarrow \mathbb{N} \times (\mathbb{N} \times \{L, S, R\})^k. \quad (2.2)$$

Finally we allow inputs and outputs of arbitrary length, turning the  $\delta_q$  into

$$\delta_q: \mathbb{N}^* \rightarrow \mathbb{N} \times (\mathbb{N} \times \{L, S, R\})^*.$$

Such a  $\delta_q$  will be called a *command*, and the elements of  $\mathbb{N} \times \{L, S, R\}$  will be called *actions*. An action consists of writing a symbol to a tape at the current tape head position and then moving the tape head.

**type-synonym**  $action = symbol \times direction$

A command maps the list of symbols read from the tapes to a follow-up state and a list of actions. It represents the machine's behavior in one state.

**type-synonym**  $command = symbol\ list \Rightarrow state \times action\ list$

Machines are then simply lists of commands. The  $q$ -th element of the list represents the machine's behavior in state  $q$ . The halting state of a machine  $M$  is *length*  $M$ , but there is obviously no such element in the list.

**type-synonym**  $machine = command\ list$

Commands in this general form are too amorphous. We call a command *well-formed* for  $k$  tapes and the state space  $Q$  if on reading  $k$  symbols it performs  $k$  actions and jumps to a state in  $\{0, \dots, Q\}$ . A well-formed command corresponds to (2.1).

**definition**  $wf\text{-}command :: nat \Rightarrow nat \Rightarrow command \Rightarrow bool$  **where**

$$wf\text{-}command\ k\ Q\ cmd \equiv \forall gs. length\ gs = k \longrightarrow length\ (snd\ (cmd\ gs)) = k \wedge fst\ (cmd\ gs) \leq Q$$

A well-formed command is a *Turing command* for  $k$  tapes and alphabet  $G$  if it writes only symbols from  $G$  when reading symbols from  $G$  and does not write to tape 0; that is, it writes to tape 0 the symbol it read from tape 0.

**definition**  $turing\text{-}command :: nat \Rightarrow nat \Rightarrow nat \Rightarrow command \Rightarrow bool$  **where**

$$\begin{aligned} turing\text{-}command\ k\ Q\ G\ cmd \equiv \\ wf\text{-}command\ k\ Q\ cmd \wedge \\ (\forall gs. length\ gs = k \longrightarrow \\ ((\forall i < k. gs\ !\ i < G) \longrightarrow (\forall i < k. fst\ (snd\ (cmd\ gs)\ !\ i) < G)) \wedge \\ (k > 0 \longrightarrow fst\ (snd\ (cmd\ gs)\ !\ 0) = gs\ !\ 0)) \end{aligned}$$

A *Turing machine* is a machine with at least two tapes and four symbols and only Turing commands.

**definition**  $turing\text{-}machine :: nat \Rightarrow nat \Rightarrow machine \Rightarrow bool$  **where**

$$turing\text{-}machine\ k\ G\ M \equiv k \geq 2 \wedge G \geq 4 \wedge (\forall cmd \in set\ M. turing\text{-}command\ k\ (length\ M)\ G\ cmd)$$

## Semantics

Next we define the semantics of machines. The state and the list of tapes make up the *configuration* of a machine. The semantics are given as functions mapping configurations to follow-up configurations.

**type-synonym**  $config = state \times tape\ list$

We start with the semantics of a single command. An action affects a tape in the following way. For the head movements we imagine the tapes having cell 0 at the left and the cell indices growing rightward.

**fun**  $act :: action \Rightarrow tape \Rightarrow tape$  **where**

$$\begin{aligned} act\ (w, m)\ tp = \\ ((fst\ tp)(snd\ tp := w), \\ case\ m\ of\ Left \Rightarrow snd\ tp - 1 \mid Stay \Rightarrow snd\ tp \mid Right \Rightarrow snd\ tp + 1) \end{aligned}$$

Reading symbols from one tape, from all tapes, and from configurations:

**abbreviation** *tape-read* :: *tape*  $\Rightarrow$  *symbol* ( $\langle | \cdot | \rangle$ ) **where**  
 $| \cdot | \equiv \text{fst } tp \text{ (snd } tp)$

**definition** *read* :: *tape list*  $\Rightarrow$  *symbol list* **where**  
 $\text{read } tps \equiv \text{map } \text{tape-read } tps$

**abbreviation** *config-read* :: *config*  $\Rightarrow$  *symbol list* **where**  
 $\text{config-read } cfg \equiv \text{read } (\text{snd } cfg)$

The semantics of a command:

**definition** *sem* :: *command*  $\Rightarrow$  *config*  $\Rightarrow$  *config* **where**  
 $\text{sem } cmd \text{ } cfg \equiv$   
 $\text{let } (\text{newstate}, \text{actions}) = cmd \text{ (config-read } cfg)$   
 $\text{in } (\text{newstate}, \text{map } (\lambda(a, tp). \text{act } a \text{ } tp) \text{ (zip actions (snd } cfg)))$

The semantics of one step of a machine consist in the semantics of the command corresponding to the state the machine is in. The following definition ensures that the configuration does not change when it is in a halting state.

**definition** *exe* :: *machine*  $\Rightarrow$  *config*  $\Rightarrow$  *config* **where**  
 $\text{exe } M \text{ } cfg \equiv \text{if } \text{fst } cfg < \text{length } M \text{ then } \text{sem } (M ! (\text{fst } cfg)) \text{ } cfg \text{ else } cfg$

Executing a machine *M* for multiple steps:

**fun** *execute* :: *machine*  $\Rightarrow$  *config*  $\Rightarrow$  *nat*  $\Rightarrow$  *config* **where**  
 $\text{execute } M \text{ } cfg \ 0 = cfg \ |$   
 $\text{execute } M \text{ } cfg \ (\text{Suc } t) = \text{exe } M \text{ (execute } M \text{ } cfg \ t)$

We have defined the semantics for arbitrary machines, but most lemmas we are going to prove about *exe*, *execute*, etc. will require the commands to be somewhat well-behaved, more precisely to map lists of *k* symbols to lists of *k* actions, as shown in (2.2). We will call such commands *proper*.

**abbreviation** *proper-command* :: *nat*  $\Rightarrow$  *command*  $\Rightarrow$  *bool* **where**  
 $\text{proper-command } k \text{ } cmd \equiv \forall gs. \text{length } gs = k \longrightarrow \text{length } (\text{snd } (cmd \text{ } gs)) = \text{length } gs$

Being proper is a weaker condition than being well-formed. Since *exe* treats the state *Q* and the states *q* > *Q* the same, we do not need the *Q*-closure property of well-formedness for most lemmas about semantics.

Next we introduce a number of abbreviations for components of a machine and aspects of its behavior. In general, symbols between bars  $| \cdot |$  represent operations on tapes, inside angle brackets  $\langle \cdot \rangle$  operations on configurations, between colons  $::$  operations on lists of tapes, and inside brackets  $[ \cdot ]$  operations on state/action-list pairs. As for the symbol inside the delimiters, a dot ( $\cdot$ ) refers to a tape symbol, a colon ( $:$ ) to the entire tape contents, and a hash ( $\#$ ) to a head position; an equals sign ( $=$ ) means some component of the left-hand side is changed. An exclamation mark (!) accesses an element in a list on the left-hand side term.

**abbreviation** *config-length* :: *config*  $\Rightarrow$  *nat* ( $\langle || - || \rangle$ ) **where**  
 $\text{config-length } cfg \equiv \text{length } (\text{snd } cfg)$

**abbreviation** *tape-move-right* :: *tape*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape* (**infixl**  $\langle | + | \rangle$  60) **where**  
 $tp \ | + | \ n \equiv (\text{fst } tp, \text{snd } tp + n)$

**abbreviation** *tape-move-left* :: *tape*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape* (**infixl**  $\langle | - | \rangle$  60) **where**  
 $tp \ | - | \ n \equiv (\text{fst } tp, \text{snd } tp - n)$

**abbreviation** *tape-move-to* :: *tape*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape* (**infixl**  $\langle | \# = | \rangle$  60) **where**  
 $tp \ | \# = | \ n \equiv (\text{fst } tp, n)$

**abbreviation** *tape-write* :: *tape*  $\Rightarrow$  *symbol*  $\Rightarrow$  *tape* (**infixl**  $\langle | := | \rangle$  60) **where**  
 $tp \ | := | \ h \equiv ((\text{fst } tp) (\text{snd } tp := h), \text{snd } tp)$

**abbreviation** *config-tape-by-no* :: *config*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape* (**infix**  $\langle \langle ! \rangle \rangle$  90) **where**

$cfg \langle ! \rangle j \equiv snd\ cfg \ ! \ j$

**abbreviation** *config-contents-by-no* ::  $config \Rightarrow nat \Rightarrow (nat \Rightarrow symbol)$  (**infix**  $\langle \langle : \rangle \rangle$  100) **where**  
 $cfg \langle \langle : \rangle \rangle j \equiv fst\ (cfg \langle ! \rangle j)$

**abbreviation** *config-pos-by-no* ::  $config \Rightarrow nat \Rightarrow nat$  (**infix**  $\langle \langle \# \rangle \rangle$  100) **where**  
 $cfg \langle \langle \# \rangle \rangle j \equiv snd\ (cfg \langle ! \rangle j)$

**abbreviation** *config-symbol-read* ::  $config \Rightarrow nat \Rightarrow symbol$  (**infix**  $\langle \langle . \rangle \rangle$  100) **where**  
 $cfg \langle . \rangle j \equiv (cfg \langle : \rangle j)\ (cfg \langle \# \rangle j)$

**abbreviation** *config-update-state* ::  $config \Rightarrow nat \Rightarrow config$  (**infix**  $\langle \langle += \rangle \rangle$  90) **where**  
 $cfg \langle += \rangle q \equiv (fst\ cfg + q, snd\ cfg)$

**abbreviation** *tapes-contents-by-no* ::  $tape\ list \Rightarrow nat \Rightarrow (nat \Rightarrow symbol)$  (**infix**  $\langle \langle :: \rangle \rangle$  100) **where**  
 $tps \langle :: \rangle j \equiv fst\ (tps \ ! \ j)$

**abbreviation** *tapes-pos-by-no* ::  $tape\ list \Rightarrow nat \Rightarrow nat$  (**infix**  $\langle \langle \# \rangle \rangle$  100) **where**  
 $tps \langle \# \rangle j \equiv snd\ (tps \ ! \ j)$

**abbreviation** *tapes-symbol-read* ::  $tape\ list \Rightarrow nat \Rightarrow symbol$  (**infix**  $\langle \langle : \rangle \rangle$  100) **where**  
 $tps \langle : \rangle j \equiv (tps \langle :: \rangle j)\ (tps \langle \# \rangle j)$

**abbreviation** *jump-by-no* ::  $state \times action\ list \Rightarrow state$  ( $\langle \langle [*] \rangle \rightarrow [90] \rangle$ ) **where**  
 $[*] \ sas \equiv fst\ sas$

**abbreviation** *actions-of-cmd* ::  $state \times action\ list \Rightarrow action\ list$  ( $\langle \langle [!] \rangle \rightarrow [100] \ 100 \rangle$ ) **where**  
 $[!] \ sas \equiv snd\ sas$

**abbreviation** *action-by-no* ::  $state \times action\ list \Rightarrow nat \Rightarrow action$  (**infix**  $\langle [!] \rangle$  90) **where**  
 $sas \ [!] \ j \equiv snd\ sas \ ! \ j$

**abbreviation** *write-by-no* ::  $state \times action\ list \Rightarrow nat \Rightarrow symbol$  (**infix**  $\langle [.] \rangle$  90) **where**  
 $sas \ [.] \ j \equiv fst\ (sas \ [!] \ j)$

**abbreviation** *direction-by-no* ::  $state \times action\ list \Rightarrow nat \Rightarrow direction$  (**infix**  $\langle [\sim] \rangle$  100) **where**  
 $sas \ [\sim] \ j \equiv snd\ (sas \ [!] \ j)$

Symbol sequences consisting of symbols from an alphabet  $G$ :

**abbreviation** *symbols-lt* ::  $nat \Rightarrow symbol\ list \Rightarrow bool$  **where**  
 $symbols\text{-}lt\ G\ rs \equiv \forall i < length\ rs. rs \ ! \ i < G$

We will frequently have to show that commands are proper or Turing commands.

**lemma** *turing-commandI* [*intro*]:

**assumes**  $\bigwedge gs. length\ gs = k \Longrightarrow length\ ([!] \ cmd\ gs) = length\ gs$   
**and**  $\bigwedge gs. length\ gs = k \Longrightarrow (\bigwedge i. i < length\ gs \Longrightarrow gs \ ! \ i < G) \Longrightarrow (\bigwedge j. j < length\ gs \Longrightarrow cmd\ gs \ [.] \ j < G)$   
**and**  $\bigwedge gs. length\ gs = k \Longrightarrow k > 0 \Longrightarrow cmd\ gs \ [.] \ 0 = gs \ ! \ 0$   
**and**  $\bigwedge gs. length\ gs = k \Longrightarrow [*] \ (cmd\ gs) \leq Q$   
**shows** *turing-command*  $k\ Q\ G\ cmd$   
 $\langle proof \rangle$

**lemma** *turing-commandD*:

**assumes** *turing-command*  $k\ Q\ G\ cmd$  **and**  $length\ gs = k$   
**shows**  $length\ ([!] \ cmd\ gs) = length\ gs$   
**and**  $(\bigwedge i. i < length\ gs \Longrightarrow gs \ ! \ i < G) \Longrightarrow (\bigwedge j. j < length\ gs \Longrightarrow cmd\ gs \ [.] \ j < G)$   
**and**  $k > 0 \Longrightarrow cmd\ gs \ [.] \ 0 = gs \ ! \ 0$   
**and**  $\bigwedge gs. length\ gs = k \Longrightarrow [*] \ (cmd\ gs) \leq Q$   
 $\langle proof \rangle$

**lemma** *turing-command-mono*:

**assumes** *turing-command*  $k\ Q\ G\ cmd$  **and**  $Q \leq Q'$   
**shows** *turing-command*  $k\ Q'\ G\ cmd$   
 $\langle proof \rangle$

**lemma** *proper-command-length*:

**assumes** *proper-command*  $k$  *cmd* **and**  $\text{length } gs = k$   
**shows**  $\text{length } ([!!] \text{ cmd } gs) = \text{length } gs$   
 $\langle \text{proof} \rangle$

**abbreviation** *proper-machine*  $:: \text{nat} \Rightarrow \text{machine} \Rightarrow \text{bool}$  **where**  
*proper-machine*  $k$   $M \equiv \forall i < \text{length } M. \text{proper-command } k (M ! i)$

**lemma** *prop-list-append*:

**assumes**  $\forall i < \text{length } M1. P (M1 ! i)$   
**and**  $\forall i < \text{length } M2. P (M2 ! i)$   
**shows**  $\forall i < \text{length } (M1 @ M2). P ((M1 @ M2) ! i)$   
 $\langle \text{proof} \rangle$

The empty Turing machine  $[]$  is the one Turing machine where the start state is the halting state, that is,  $q_{start} = q_{halt} = Q = 0$ . It is a Turing machine for every  $k \geq 2$  and  $G \geq 4$ :

**lemma** *Nil-tm*:  $G \geq 4 \Longrightarrow k \geq 2 \Longrightarrow \text{turing-machine } k$   $G$   $[]$   
 $\langle \text{proof} \rangle$

**lemma** *turing-machineI* [*intro*]:

**assumes**  $k \geq 2$   
**and**  $G \geq 4$   
**and**  $\bigwedge i. i < \text{length } M \Longrightarrow \text{turing-command } k (\text{length } M) G (M ! i)$   
**shows** *turing-machine*  $k$   $G$   $M$   
 $\langle \text{proof} \rangle$

**lemma** *turing-machineD*:

**assumes** *turing-machine*  $k$   $G$   $M$   
**shows**  $k \geq 2$   
**and**  $G \geq 4$   
**and**  $\bigwedge i. i < \text{length } M \Longrightarrow \text{turing-command } k (\text{length } M) G (M ! i)$   
 $\langle \text{proof} \rangle$

A few lemmas about *act*, *read*, and *sem*:

**lemma** *act*:  $\text{act } a \text{ tp} =$

$((\text{fst } \text{tp})(\text{snd } \text{tp} := \text{fst } a),$   
 $\text{case } \text{snd } a \text{ of } \text{Left} \Rightarrow \text{snd } \text{tp} - 1 \mid \text{Stay} \Rightarrow \text{snd } \text{tp} \mid \text{Right} \Rightarrow \text{snd } \text{tp} + 1)$   
 $\langle \text{proof} \rangle$

**lemma** *act-Stay*:  $j < \text{length } \text{tps} \Longrightarrow \text{act } (\text{read } \text{tps} ! j, \text{Stay}) (\text{tps} ! j) = \text{tps} ! j$   
 $\langle \text{proof} \rangle$

**lemma** *act-Right*:  $j < \text{length } \text{tps} \Longrightarrow \text{act } (\text{read } \text{tps} ! j, \text{Right}) (\text{tps} ! j) = \text{tps} ! j \mid + \mid 1$   
 $\langle \text{proof} \rangle$

**lemma** *act-Left*:  $j < \text{length } \text{tps} \Longrightarrow \text{act } (\text{read } \text{tps} ! j, \text{Left}) (\text{tps} ! j) = \text{tps} ! j \mid - \mid 1$   
 $\langle \text{proof} \rangle$

**lemma** *act-Stay'*:  $\text{act } (h, \text{Stay}) (\text{tps} ! j) = \text{tps} ! j \mid := \mid h$   
 $\langle \text{proof} \rangle$

**lemma** *act-Right'*:  $\text{act } (h, \text{Right}) (\text{tps} ! j) = \text{tps} ! j \mid := \mid h \mid + \mid 1$   
 $\langle \text{proof} \rangle$

**lemma** *act-Left'*:  $\text{act } (h, \text{Left}) (\text{tps} ! j) = \text{tps} ! j \mid := \mid h \mid - \mid 1$   
 $\langle \text{proof} \rangle$

**lemma** *act-pos-le-Suc*:  $\text{snd } (\text{act } a (\text{tps} ! j)) \leq \text{Suc } (\text{snd } (\text{tps} ! j))$   
 $\langle \text{proof} \rangle$

**lemma** *act-changes-at-most-pos*:

**assumes**  $i \neq \text{snd } tp$   
**shows**  $\text{fst } (\text{act } (h, mv) \text{ } tp) \ i = \text{fst } tp \ i$   
 $\langle \text{proof} \rangle$

**lemma** *act-changes-at-most-pos'*:

**assumes**  $i \neq \text{snd } tp$   
**shows**  $\text{fst } (\text{act } a \text{ } tp) \ i = \text{fst } tp \ i$   
 $\langle \text{proof} \rangle$

**lemma** *read-length*:  $\text{length } (\text{read } tps) = \text{length } tps$   
 $\langle \text{proof} \rangle$

**lemma** *tapes-at-read*:  $j < \text{length } tps \implies (q, tps) <.> j = \text{read } tps \ ! \ j$   
 $\langle \text{proof} \rangle$

**lemma** *tapes-at-read'*:  $j < \text{length } tps \implies tps \ :: \ j = \text{read } tps \ ! \ j$   
 $\langle \text{proof} \rangle$

**lemma** *read-abbrev*:  $j < \|\text{cfg}\| \implies \text{read } (\text{snd } \text{cfg}) \ ! \ j = \text{cfg} \ <.> \ j$   
 $\langle \text{proof} \rangle$

**lemma** *sem*:

$\text{sem } \text{cmd } \text{cfg} =$   
 $(\text{let } rs = \text{read } (\text{snd } \text{cfg})$   
 $\text{in } (\text{fst } (\text{cmd } rs), \text{map } (\lambda(a, tp). \text{act } a \text{ } tp) (\text{zip } (\text{snd } (\text{cmd } rs)) (\text{snd } \text{cfg}))))$   
 $\langle \text{proof} \rangle$

**lemma** *sem'*:

$\text{sem } \text{cmd } \text{cfg} =$   
 $(\text{fst } (\text{cmd } (\text{read } (\text{snd } \text{cfg}))), \text{map } (\lambda(a, tp). \text{act } a \text{ } tp) (\text{zip } (\text{snd } (\text{cmd } (\text{read } (\text{snd } \text{cfg})))) (\text{snd } \text{cfg}))))$   
 $\langle \text{proof} \rangle$

**lemma** *sem''*:

$\text{sem } \text{cmd } (q, tps) =$   
 $(\text{fst } (\text{cmd } (\text{read } tps)), \text{map } (\lambda(a, tp). \text{act } a \text{ } tp) (\text{zip } (\text{snd } (\text{cmd } (\text{read } tps))) tps))$   
 $\langle \text{proof} \rangle$

**lemma** *sem-num-tapes-raw*:  $\text{proper-command } k \ \text{cmd} \implies k = \|\text{cfg}\| \implies k = \|\text{sem } \text{cmd } \text{cfg}\|$   
 $\langle \text{proof} \rangle$

**lemma** *sem-num-tapes2*:  $\text{turing-command } k \ Q \ G \ \text{cmd} \implies k = \|\text{cfg}\| \implies k = \|\text{sem } \text{cmd } \text{cfg}\|$   
 $\langle \text{proof} \rangle$

**corollary** *sem-num-tapes2'*:  $\text{turing-command } \|\text{cfg}\| \ Q \ G \ \text{cmd} \implies \|\text{cfg}\| = \|\text{sem } \text{cmd } \text{cfg}\|$   
 $\langle \text{proof} \rangle$

**corollary** *sem-num-tapes3*:  $\text{turing-command } \|\text{cfg}\| \ Q \ G \ \text{cmd} \implies \|\text{cfg}\| = \|\text{sem } \text{cmd } \text{cfg}\|$   
 $\langle \text{proof} \rangle$

**lemma** *sem-fst*:

**assumes**  $\text{cfg}' = \text{sem } \text{cmd } \text{cfg}$  **and**  $rs = \text{read } (\text{snd } \text{cfg})$   
**shows**  $\text{fst } \text{cfg}' = \text{fst } (\text{cmd } rs)$   
 $\langle \text{proof} \rangle$

**lemma** *sem-snd*:

**assumes** *proper-command*  $k \ \text{cmd}$   
**and**  $\|\text{cfg}\| = k$   
**and**  $rs = \text{read } (\text{snd } \text{cfg})$   
**and**  $j < k$   
**shows**  $\text{sem } \text{cmd } \text{cfg} \ <!\> \ j = \text{act } (\text{snd } (\text{cmd } rs) \ ! \ j) \ (\text{snd } \text{cfg} \ ! \ j)$   
 $\langle \text{proof} \rangle$

**lemma** *snd-semI*:

**assumes** *proper-command*  $k$   $cmd$   
**and**  $length\ tps = k$   
**and**  $length\ tps' = k$   
**and**  $\bigwedge j. j < k \implies act\ (cmd\ (read\ tps)\ [!]\ j)\ (tps\ !\ j) = tps'\ !\ j$   
**shows**  $snd\ (sem\ cmd\ (q,\ tps)) = snd\ (q',\ tps')$   
 $\langle proof \rangle$

**lemma** *sem-snd-tm*:

**assumes** *turing-machine*  $k$   $G$   $M$   
**and**  $length\ tps = k$   
**and**  $rs = read\ tps$   
**and**  $j < k$   
**and**  $q < length\ M$   
**shows**  $sem\ (M\ !\ q)\ (q,\ tps)\ <!\>\ j = act\ (snd\ ((M\ !\ q)\ rs)\ !\ j)\ (tps\ !\ j)$   
 $\langle proof \rangle$

**lemma** *semI*:

**assumes** *proper-command*  $k$   $cmd$   
**and**  $length\ tps = k$   
**and**  $length\ tps' = k$   
**and**  $fst\ (cmd\ (read\ tps)) = q'$   
**and**  $\bigwedge j. j < k \implies act\ (cmd\ (read\ tps)\ [!]\ j)\ (tps\ !\ j) = tps'\ !\ j$   
**shows**  $sem\ cmd\ (q,\ tps) = (q',\ tps')$   
 $\langle proof \rangle$

Commands ignore the state element of the configuration they are applied to.

**lemma** *sem-state-indep*:

**assumes**  $snd\ cfg1 = snd\ cfg2$   
**shows**  $sem\ cmd\ cfg1 = sem\ cmd\ cfg2$   
 $\langle proof \rangle$

A few lemmas about *exe* and *execute*:

**lemma** *exe-lt-length*:  $fst\ cfg < length\ M \implies exe\ M\ cfg = sem\ (M\ !\ (fst\ cfg))\ cfg$   
 $\langle proof \rangle$

**lemma** *exe-ge-length*:  $fst\ cfg \geq length\ M \implies exe\ M\ cfg = cfg$   
 $\langle proof \rangle$

**lemma** *exe-num-tapes*:

**assumes** *turing-machine*  $k$   $G$   $M$  **and**  $k = ||cfg||$   
**shows**  $k = ||exe\ M\ cfg||$   
 $\langle proof \rangle$

**lemma** *exe-num-tapes-proper*:

**assumes** *proper-machine*  $k$   $M$  **and**  $k = ||cfg||$   
**shows**  $k = ||exe\ M\ cfg||$   
 $\langle proof \rangle$

**lemma** *execute-num-tapes-proper*:

**assumes** *proper-machine*  $k$   $M$  **and**  $k = ||cfg||$   
**shows**  $k = ||execute\ M\ cfg\ t||$   
 $\langle proof \rangle$

**lemma** *execute-num-tapes*:

**assumes** *turing-machine*  $k$   $G$   $M$  **and**  $k = ||cfg||$   
**shows**  $k = ||execute\ M\ cfg\ t||$   
 $\langle proof \rangle$

**lemma** *execute-after-halting*:

**assumes**  $fst\ (execute\ M\ cfg0\ t) = length\ M$   
**shows**  $execute\ M\ cfg0\ (t + n) = execute\ M\ cfg0\ t$   
 $\langle proof \rangle$

**lemma** *execute-after-halting'*:  
**assumes**  $\text{fst } (\text{execute } M \text{ cfg0 } t) \geq \text{length } M$   
**shows**  $\text{execute } M \text{ cfg0 } (t + n) = \text{execute } M \text{ cfg0 } t$   
 $\langle \text{proof} \rangle$

**corollary** *execute-after-halting-ge*:  
**assumes**  $\text{fst } (\text{execute } M \text{ cfg0 } t) = \text{length } M$  **and**  $t \leq t'$   
**shows**  $\text{execute } M \text{ cfg0 } t' = \text{execute } M \text{ cfg0 } t$   
 $\langle \text{proof} \rangle$

**corollary** *execute-after-halting-ge'*:  
**assumes**  $\text{fst } (\text{execute } M \text{ cfg0 } t) \geq \text{length } M$  **and**  $t \leq t'$   
**shows**  $\text{execute } M \text{ cfg0 } t' = \text{execute } M \text{ cfg0 } t$   
 $\langle \text{proof} \rangle$

**lemma** *execute-additive*:  
**assumes**  $\text{execute } M \text{ cfg1 } t1 = \text{cfg2}$  **and**  $\text{execute } M \text{ cfg2 } t2 = \text{cfg3}$   
**shows**  $\text{execute } M \text{ cfg1 } (t1 + t2) = \text{cfg3}$   
 $\langle \text{proof} \rangle$

**lemma** *turing-machine-execute-states*:  
**assumes** *turing-machine*  $k \ G \ M$  **and**  $\text{fst } \text{cfg} \leq \text{length } M$  **and**  $\|\text{cfg}\| = k$   
**shows**  $\text{fst } (\text{execute } M \text{ cfg } t) \leq \text{length } M$   
 $\langle \text{proof} \rangle$

While running times are important, usually upper bounds for them suffice. The next predicate expresses that a machine *transits* from one configuration to another one in at most a certain number of steps.

**definition** *transits* :: *machine*  $\Rightarrow$  *config*  $\Rightarrow$  *nat*  $\Rightarrow$  *config*  $\Rightarrow$  *bool* **where**  
 $\text{transits } M \text{ cfg1 } t \text{ cfg2} \equiv \exists t' \leq t. \text{execute } M \text{ cfg1 } t' = \text{cfg2}$

**lemma** *transits-monotone*:  
**assumes**  $t \leq t'$  **and**  $\text{transits } M \text{ cfg1 } t \text{ cfg2}$   
**shows**  $\text{transits } M \text{ cfg1 } t' \text{ cfg2}$   
 $\langle \text{proof} \rangle$

**lemma** *transits-additive*:  
**assumes**  $\text{transits } M \text{ cfg1 } t1 \text{ cfg2}$  **and**  $\text{transits } M \text{ cfg2 } t2 \text{ cfg3}$   
**shows**  $\text{transits } M \text{ cfg1 } (t1 + t2) \text{ cfg3}$   
 $\langle \text{proof} \rangle$

**lemma** *transitsI*:  
**assumes**  $\text{execute } M \text{ cfg1 } t' = \text{cfg2}$  **and**  $t' \leq t$   
**shows**  $\text{transits } M \text{ cfg1 } t \text{ cfg2}$   
 $\langle \text{proof} \rangle$

**lemma** *execute-imp-transits*:  
**assumes**  $\text{execute } M \text{ cfg1 } t = \text{cfg2}$   
**shows**  $\text{transits } M \text{ cfg1 } t \text{ cfg2}$   
 $\langle \text{proof} \rangle$

In the vast majority of cases we are only interested in transitions from the start state to the halting state. One way to look at it is the machine *transforms* a list of tapes to another list of tapes within a certain number of steps.

**definition** *transforms* :: *machine*  $\Rightarrow$  *tape list*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape list*  $\Rightarrow$  *bool* **where**  
 $\text{transforms } M \text{ tps } t \text{ tps}' \equiv \text{transits } M (0, \text{tps}) t (\text{length } M, \text{tps}'^{\wedge})$

The previous predicate will be the standard way in which we express the behavior of a (Turing) machine. Consider, for example, the empty machine:

**lemma** *transforms-Nil*:  $\text{transforms } [] \text{ tps } 0 \text{ tps}$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-monotone*:

**assumes** *transforms*  $M$  *tps*  $t$  *tps'* **and**  $t \leq t'$   
**shows** *transforms*  $M$  *tps*  $t'$  *tps'*  
 ⟨*proof*⟩

Most often the tapes will have a start symbol in the first cell followed by a finite sequence of symbols.

**definition** *contents* :: *symbol list*  $\Rightarrow$  ( $\text{nat} \Rightarrow \text{symbol}$ ) ( $\langle [-] \rangle$ ) **where**  
 $[xs] i \equiv \text{if } i = 0 \text{ then } \triangleright \text{ else if } i \leq \text{length } xs \text{ then } xs ! (i - 1) \text{ else } \square$

**lemma** *contents-at-0* [*simp*]:  $[zs] 0 = \triangleright$   
 ⟨*proof*⟩

**lemma** *contents-inbounds* [*simp*]:  $i > 0 \implies i \leq \text{length } zs \implies [zs] i = zs ! (i - 1)$   
 ⟨*proof*⟩

**lemma** *contents-outofbounds* [*simp*]:  $i > \text{length } zs \implies [zs] i = \square$   
 ⟨*proof*⟩

When Turing machines are used to compute functions, they are started in a specific configuration where all tapes have the format just defined and the first tape contains the input. This is called the *start configuration* [2, p. 13].

**definition** *start-config* ::  $\text{nat} \Rightarrow \text{symbol list} \Rightarrow \text{config}$  **where**  
 $\text{start-config } k \text{ } xs \equiv (0, ([xs], 0) \# \text{replicate } (k - 1) ([\square], 0))$

**lemma** *start-config-length*:  $k > 0 \implies \|\text{start-config } k \text{ } xs\| = k$   
 ⟨*proof*⟩

**lemma** *start-config1*:  
**assumes**  $\text{cfg} = \text{start-config } k \text{ } xs$  **and**  $0 < j$  **and**  $j < k$  **and**  $i > 0$   
**shows**  $(\text{cfg} <:> j) i = \square$   
 ⟨*proof*⟩

**lemma** *start-config2*:  
**assumes**  $\text{cfg} = \text{start-config } k \text{ } xs$  **and**  $j < k$   
**shows**  $(\text{cfg} <:> j) 0 = \triangleright$   
 ⟨*proof*⟩

**lemma** *start-config3*:  
**assumes**  $\text{cfg} = \text{start-config } k \text{ } xs$  **and**  $i > 0$  **and**  $i \leq \text{length } xs$   
**shows**  $(\text{cfg} <:> 0) i = xs ! (i - 1)$   
 ⟨*proof*⟩

**lemma** *start-config4*:  
**assumes**  $0 < j$  **and**  $j < k$   
**shows**  $\text{snd } (\text{start-config } k \text{ } xs) ! j = (\lambda i. \text{if } i = 0 \text{ then } \triangleright \text{ else } \square, 0)$   
 ⟨*proof*⟩

**lemma** *start-config-pos*:  $j < k \implies \text{start-config } k \text{ } zs <:\#> j = 0$   
 ⟨*proof*⟩

We call a symbol *proper* if it is neither the blank symbol nor the start symbol.

**abbreviation** *proper-symbols* :: *symbol list*  $\Rightarrow$  *bool* **where**  
 $\text{proper-symbols } xs \equiv \forall i < \text{length } xs. xs ! i > \text{Suc } 0$

**lemma** *proper-symbols-append*:  
**assumes** *proper-symbols*  $xs$  **and** *proper-symbols*  $ys$   
**shows** *proper-symbols*  $(xs @ ys)$   
 ⟨*proof*⟩

**lemma** *proper-symbols-ne0*: *proper-symbols*  $xs \implies \forall i < \text{length } xs. xs ! i \neq \square$   
 ⟨*proof*⟩

**lemma** *proper-symbols-ne1*: *proper-symbols*  $xs \implies \forall i < \text{length } xs. xs ! i \neq \triangleright$   
 ⟨*proof*⟩

We call the symbols **0** and **1** *bit symbols*.

**abbreviation** *bit-symbols* :: nat list  $\Rightarrow$  bool **where**  
*bit-symbols* xs  $\equiv \forall i < \text{length } xs. xs ! i = \mathbf{0} \vee xs ! i = \mathbf{1}$

**lemma** *bit-symbols-append*:  
**assumes** *bit-symbols* xs **and** *bit-symbols* ys  
**shows** *bit-symbols* (xs @ ys)  
 <proof>

## Basic facts about Turing machines

A Turing machine with alphabet  $G$  started on a symbol sequence over  $G$  will only ever have symbols from  $G$  on any of its tapes.

**lemma** *tape-alphabet*:  
**assumes** *turing-machine* k G M **and** *symbols-lt* G zs **and**  $j < k$   
**shows** ((*execute* M (*start-config* k zs) t) <:> j) i < G  
 <proof>

**corollary** *read-alphabet*:  
**assumes** *turing-machine* k G M **and** *symbols-lt* G zs  
**shows**  $\forall i < k. \text{config-read } (\text{execute } M (\text{start-config } k \text{ zs}) t) ! i < G$   
 <proof>

**corollary** *read-alphabet'*:  
**assumes** *turing-machine* k G M **and** *symbols-lt* G zs  
**shows** *symbols-lt* G (*config-read* (*execute* M (*start-config* k zs) t))  
 <proof>

**corollary** *read-alphabet-set*:  
**assumes** *turing-machine* k G M **and** *symbols-lt* G zs  
**shows**  $\forall h \in \text{set } (\text{config-read } (\text{execute } M (\text{start-config } k \text{ zs}) t)). h < G$   
 <proof>

The contents of the input tape never change.

**lemma** *input-tape-constant*:  
**assumes** *turing-machine* k G M **and**  $k = \|\text{cfg}\|$   
**shows** *execute* M cfg t <:> 0 = *execute* M cfg 0 <:> 0  
 <proof>

A head position cannot be greater than the number of steps the machine has been running.

**lemma** *head-pos-le-time*:  
**assumes** *turing-machine* k G M **and**  $j < k$   
**shows** *execute* M (*start-config* k zs) t <#>  $j \leq t$   
 <proof>

**lemma** *head-pos-le-halting-time*:  
**assumes** *turing-machine* k G M  
**and** *fst* (*execute* M (*start-config* k zs) T) = *length* M  
**and**  $j < k$   
**shows** *execute* M (*start-config* k zs) t <#>  $j \leq T$   
 <proof>

A tape cannot contain non-blank symbols at a position larger than the number of steps the Turing machine has been running, except on the input tape.

**lemma** *blank-after-time*:  
**assumes**  $i > t$  **and**  $j < k$  **and**  $0 < j$  **and** *turing-machine* k G M  
**shows** (*execute* M (*start-config* k zs) t <:> j) i =  $\square$   
 <proof>

## 2.1.2 Computing a function

Turing machines are supposed to compute functions. The functions in question map bit strings to bit strings. We model such strings as lists of Booleans and denote the bits by **O** and **I**.

**type-synonym**  $string = bool\ list$

**notation**  $False$  ( $\langle \mathbf{O} \rangle$ ) and  $True$  ( $\langle \mathbf{I} \rangle$ )

This keeps the more abstract level of computable functions separate from the level of concrete implementations as Turing machines, which can use an arbitrary alphabet. We use the term “string” only for bit strings, on which functions operate, and the terms “symbol sequence” or “symbols” for the things written on the tapes of Turing machines. We translate between the two levels in a straightforward way:

**abbreviation**  $string\text{-}to\text{-}symbols :: string \Rightarrow symbol\ list$  **where**  
 $string\text{-}to\text{-}symbols\ x \equiv map\ (\lambda b.\ if\ b\ then\ \mathbf{1}\ else\ \mathbf{0})\ x$

**abbreviation**  $symbols\text{-}to\text{-}string :: symbol\ list \Rightarrow string$  **where**  
 $symbols\text{-}to\text{-}string\ zs \equiv map\ (\lambda z.\ z = \mathbf{1})\ zs$

**proposition**

$string\text{-}to\text{-}symbols\ [\mathbf{O}, \mathbf{I}] = [\mathbf{0}, \mathbf{1}]$   
 $symbols\text{-}to\text{-}string\ [\mathbf{0}, \mathbf{1}] = [\mathbf{O}, \mathbf{I}]$   
 $\langle proof \rangle$

**lemma**  $bit\text{-}symbols\text{-}to\text{-}symbols$ :

**assumes**  $bit\text{-}symbols\ zs$   
**shows**  $string\text{-}to\text{-}symbols\ (symbols\text{-}to\text{-}string\ zs) = zs$   
 $\langle proof \rangle$

**lemma**  $symbols\text{-}to\text{-}string\text{-}to\text{-}symbols$ :  $symbols\text{-}to\text{-}string\ (string\text{-}to\text{-}symbols\ x) = x$   
 $\langle proof \rangle$

**lemma**  $proper\text{-}symbols\text{-}to\text{-}symbols$ :  $proper\text{-}symbols\ (string\text{-}to\text{-}symbols\ zs)$   
 $\langle proof \rangle$

**abbreviation**  $string\text{-}to\text{-}contents :: string \Rightarrow (nat \Rightarrow symbol)$  **where**  
 $string\text{-}to\text{-}contents\ x \equiv$   
 $\lambda i.\ if\ i = 0\ then\ \triangleright\ else\ if\ i \leq length\ x\ then\ (if\ x\ !\ (i - 1)\ then\ \mathbf{1}\ else\ \mathbf{0})\ else\ \square$

**lemma**  $contents\text{-}string\text{-}to\text{-}contents$ :  $string\text{-}to\text{-}contents\ xs = \lfloor string\text{-}to\text{-}symbols\ xs \rfloor$   
 $\langle proof \rangle$

**lemma**  $bit\text{-}symbols\text{-}to\text{-}contents$ :

**assumes**  $bit\text{-}symbols\ ns$   
**shows**  $\lfloor ns \rfloor = string\text{-}to\text{-}contents\ (symbols\text{-}to\text{-}string\ ns)$   
 $\langle proof \rangle$

Definition 1.3 in the textbook [2] says that for a Turing machine  $M$  to compute a function  $f: \{\mathbf{O}, \mathbf{I}\}^* \rightarrow \{\mathbf{O}, \mathbf{I}\}^*$  on input  $x$ , “it halts with  $f(x)$  written on its output tape.” My initial interpretation of this phrase, and the one formalized below, was that the output is written *after* the start symbol  $\triangleright$  in the same fashion as the input is given on the input tape. However after inspecting the Turing machine in Example 1.1, I now believe the more likely meaning is that the output *overwrites* the start symbol, although Example 1.1 precedes Definition 1.3 and might not be subject to it.

One advantage of the interpretation with start symbol intact is that the output tape can then be used unchanged as the input of another Turing machine, a property we exploit in Section 2.6. Otherwise one would have to find the start cell of the output tape and either copy the contents to another tape with start symbol or shift the string to the right and restore the start symbol. One way to find the start cell is to move the tape head left while “marking” the cells until one reaches an already marked cell, which can only happen when the head is in the start cell, where “moving left” does not actually move the head. This process will take time linear in the length of the output and thus will not change the asymptotic running time of the machine. Therefore the choice of interpretation is purely one of convenience.

**definition** *halts* :: *machine*  $\Rightarrow$  *config*  $\Rightarrow$  *bool* **where**  
*halts* *M cfg*  $\equiv \exists t. \text{fst} (\text{execute } M \text{ cfg } t) = \text{length } M$

**lemma** *halts-impl-le-length*:  
**assumes** *halts* *M cfg*  
**shows** *fst* (*execute* *M cfg t*)  $\leq$  *length* *M*  
*<proof>*

**definition** *running-time* :: *machine*  $\Rightarrow$  *config*  $\Rightarrow$  *nat* **where**  
*running-time* *M cfg*  $\equiv \text{LEAST } t. \text{fst} (\text{execute } M \text{ cfg } t) = \text{length } M$

**lemma** *running-timeD*:  
**assumes** *running-time* *M cfg* = *t* **and** *halts* *M cfg*  
**shows** *fst* (*execute* *M cfg t*) = *length* *M*  
**and**  $\bigwedge t'. t' < t \implies \text{fst} (\text{execute } M \text{ cfg } t') \neq \text{length } M$   
*<proof>*

**definition** *halting-config* :: *machine*  $\Rightarrow$  *config*  $\Rightarrow$  *config* **where**  
*halting-config* *M cfg*  $\equiv \text{execute } M \text{ cfg } (\text{running-time } M \text{ cfg})$

**abbreviation** *start-config-string* :: *nat*  $\Rightarrow$  *string*  $\Rightarrow$  *config* **where**  
*start-config-string* *k x*  $\equiv \text{start-config } k (\text{string-to-symbols } x)$

Another, inconsequential, difference to the textbook definition is that we designate the second tape, rather than the last tape, as the output tape. This means that the indices for the input and output tape are fixed at 0 and 1, respectively, regardless of the total number of tapes. Next is our definition of a *k*-tape Turing machine *M* computing a function *f* in *T*-time:

**definition** *computes-in-time* :: *nat*  $\Rightarrow$  *machine*  $\Rightarrow$  (*string*  $\Rightarrow$  *string*)  $\Rightarrow$  (*nat*  $\Rightarrow$  *nat*)  $\Rightarrow$  *bool* **where**  
*computes-in-time* *k M f T*  $\equiv \forall x.$   
*halts* *M* (*start-config-string* *k x*)  $\wedge$   
*running-time* *M* (*start-config-string* *k x*)  $\leq T (\text{length } x) \wedge$   
*halting-config* *M* (*start-config-string* *k x*)  $\langle : \rangle 1 = \text{string-to-contents } (f x)$

**lemma** *computes-in-time-mono*:  
**assumes** *computes-in-time* *k M f T* **and**  $\bigwedge n. T n \leq T' n$   
**shows** *computes-in-time* *k M f T'*  
*<proof>*

The definition of *computes-in-time* can be expressed with *transforms* as well, which will be more convenient for us.

**lemma** *halting-config-execute*:  
**assumes** *fst* (*execute* *M cfg t*) = *length* *M*  
**shows** *halting-config* *M cfg* = *execute* *M cfg t*  
*<proof>*

**lemma** *transforms-halting-config*:  
**assumes** *transforms* *M tps t tps'*  
**shows** *halting-config* *M* (*0, tps*) = (*length* *M, tps'*)  
*<proof>*

**lemma** *computes-in-time-execute*:  
**assumes** *computes-in-time* *k M f T*  
**shows** *execute* *M* (*start-config-string* *k x*) (*T* (*length* *x*))  $\langle : \rangle 1 = \text{string-to-contents } (f x)$   
*<proof>*

**lemma** *transforms-running-time*:  
**assumes** *transforms* *M tps t tps'*  
**shows** *running-time* *M* (*0, tps*)  $\leq t$   
*<proof>*

This is the alternative characterization of *computes-in-time*:

**lemma** *computes-in-time-alt*:

*computes-in-time*  $k$   $M$   $f$   $T =$   
 $(\forall x. \exists tps.$   
 $tps ::: 1 = \text{string-to-contents } (f\ x) \wedge$   
 $\text{transforms } M (\text{snd } (\text{start-config-string } k\ x)) (T (\text{length } x))\ tps)$   
 $(\text{is } ?lhs = ?rhs)$   
 $\langle \text{proof} \rangle$

**lemma** *computes-in-timeD*:  
**fixes**  $x$   
**assumes** *computes-in-time*  $k$   $M$   $f$   $T$   
**shows**  $\exists tps. tps ::: 1 = \text{string-to-contents } (f\ x) \wedge$   
 $\text{transforms } M (\text{snd } (\text{start-config } k (\text{string-to-symbols } x))) (T (\text{length } x))\ tps$   
 $\langle \text{proof} \rangle$

**lemma** *computes-in-timeI* [intro]:  
**assumes**  $\bigwedge x. \exists tps. tps ::: 1 = \text{string-to-contents } (f\ x) \wedge$   
 $\text{transforms } M (\text{snd } (\text{start-config } k (\text{string-to-symbols } x))) (T (\text{length } x))\ tps$   
**shows** *computes-in-time*  $k$   $M$   $f$   $T$   
 $\langle \text{proof} \rangle$

As an example, the function mapping every string to the empty string is computable within any time bound by the empty Turing machine.

**lemma** *computes-Nil-empty*:  
**assumes**  $k \geq 2$   
**shows** *computes-in-time*  $k$   $[]$   $(\lambda x. [])$   $T$   
 $\langle \text{proof} \rangle$

### 2.1.3 Pairing strings

In order to define the computability of functions with two arguments, we need a way to encode a pair of strings as one string. The idea is to write the two strings with a separator, for example, **0100#1110** and then encode every symbol **0**, **1**, **#** by two bits from  $\{0, 1\}$ . We slightly deviate from Arora and Barak's encoding [2, p. 2] and map **0** to **00**, **1** to **01**, and **#** to **11**, the idea being that the first bit signals whether the second bit is to be taken literally or as a special character. Our example turns into **000100001101010100**.

**abbreviation** *bitenc* :: *string*  $\Rightarrow$  *string* **where**  
 $\text{bitenc } x \equiv \text{concat } (\text{map } (\lambda h. [0, h])\ x)$

**definition** *string-pair* :: *string*  $\Rightarrow$  *string*  $\Rightarrow$  *string*  $\langle \langle -, - \rangle \rangle$  **where**  
 $\langle x, y \rangle \equiv \text{bitenc } x @ [1, 1] @ \text{bitenc } y$

Our example:

**proposition**  $\langle [0, 1, 0, 0], [1, 1, 1, 0] \rangle = [0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0]$   
 $\langle \text{proof} \rangle$

**lemma** *length-string-pair*:  $\text{length } \langle x, y \rangle = 2 * \text{length } x + 2 * \text{length } y + 2$   
 $\langle \text{proof} \rangle$

**lemma** *length-bitenc*:  $\text{length } (\text{bitenc } z) = 2 * \text{length } z$   
 $\langle \text{proof} \rangle$

**lemma** *bitenc-nth*:  
**assumes**  $i < \text{length } zs$   
**shows**  $\text{bitenc } zs ! (2 * i) = 0$   
**and**  $\text{bitenc } zs ! (2 * i + 1) = zs ! i$   
 $\langle \text{proof} \rangle$

**lemma** *string-pair-first-nth*:  
**assumes**  $i < \text{length } x$   
**shows**  $\langle x, y \rangle ! (2 * i) = 0$   
**and**  $\langle x, y \rangle ! (2 * i + 1) = x ! i$

*<proof>*

**lemma** *string-pair-sep-nth*:

**shows**  $\langle x, y \rangle ! (2 * \text{length } x) = \mathbf{I}$   
**and**  $\langle x, y \rangle ! (2 * \text{length } x + 1) = \mathbf{I}$   
*<proof>*

**lemma** *string-pair-second-nth*:

**assumes**  $i < \text{length } y$   
**shows**  $\langle x, y \rangle ! (2 * \text{length } x + 2 + 2 * i) = \mathbf{O}$   
**and**  $\langle x, y \rangle ! (2 * \text{length } x + 2 + 2 * i + 1) = y ! i$   
*<proof>*

**lemma** *string-pair-inj*:

**assumes**  $\langle x1, y1 \rangle = \langle x2, y2 \rangle$   
**shows**  $x1 = x2 \wedge y1 = y2$   
*<proof>*

Turing machines have to deal with pairs of symbol sequences rather than strings.

**abbreviation** *pair* :: *string*  $\Rightarrow$  *string*  $\Rightarrow$  *symbol list* ( $\langle \langle -; - \rangle \rangle$ ) **where**  
 $\langle x; y \rangle \equiv \text{string-to-symbols } \langle x, y \rangle$

**lemma** *symbols-lt-pair*: *symbols-lt* 4  $\langle x; y \rangle$   
*<proof>*

**lemma** *length-pair*: *length*  $\langle x; y \rangle = 2 * \text{length } x + 2 * \text{length } y + 2$   
*<proof>*

**lemma** *pair-inj*:

**assumes**  $\langle x1; y1 \rangle = \langle x2; y2 \rangle$   
**shows**  $x1 = x2 \wedge y1 = y2$   
*<proof>*

## 2.1.4 Big-Oh and polynomials

The Big-Oh notation is standard [2, Definition 0.2]. It can be defined with  $c$  ranging over real or natural numbers. We choose natural numbers for simplicity.

**definition** *big-oh* :: (*nat*  $\Rightarrow$  *nat*)  $\Rightarrow$  (*nat*  $\Rightarrow$  *nat*)  $\Rightarrow$  *bool* **where**  
 $\text{big-oh } g f \equiv \exists c m. \forall n > m. g n \leq c * f n$

Some examples:

**proposition** *big-oh*  $(\lambda n. n)$   $(\lambda n. n)$   
*<proof>*

**proposition** *big-oh*  $(\lambda n. n)$   $(\lambda n. n * n)$   
*<proof>*

**proposition** *big-oh*  $(\lambda n. 42 * n)$   $(\lambda n. n * n)$   
*<proof>*

**proposition**  $\neg \text{big-oh } (\lambda n. n * n)$   $(\lambda n. n)$  (**is**  $\neg \text{big-oh } ?g ?f$ )  
*<proof>*

Some lemmas helping with polynomial upper bounds.

**lemma** *pow-mono*:

**fixes**  $n d1 d2 :: \text{nat}$   
**assumes**  $d1 \leq d2$  **and**  $n > 0$   
**shows**  $n \wedge d1 \leq n \wedge d2$   
*<proof>*

**lemma** *pow-mono'*:

**fixes**  $n d1 d2 :: \text{nat}$

**assumes**  $d1 \leq d2$  **and**  $0 < d1$   
**shows**  $n \wedge d1 \leq n \wedge d2$   
 ⟨proof⟩

**lemma** *linear-le-pow*:  
**fixes**  $n d1 :: nat$   
**assumes**  $0 < d1$   
**shows**  $n \leq n \wedge d1$   
 ⟨proof⟩

The next definition formalizes the phrase “polynomially bounded” and the term “polynomial” in “polynomial running-time”. This is often written “ $f(n) = n^{O(1)}$ ” (for example, Arora and Barak [2, Example 0.3]).

**definition** *big-oh-poly* ::  $(nat \Rightarrow nat) \Rightarrow bool$  **where**  
*big-oh-poly*  $f \equiv \exists d. big\text{-}oh\ f\ (\lambda n. n \wedge d)$

**lemma** *big-oh-poly*: *big-oh-poly*  $f \longleftrightarrow (\exists d\ c\ n_0. \forall n > n_0. f\ n \leq c * n \wedge d)$   
 ⟨proof⟩

**lemma** *big-oh-polyI*:  
**assumes**  $\bigwedge n. n > n_0 \implies f\ n \leq c * n \wedge d$   
**shows** *big-oh-poly*  $f$   
 ⟨proof⟩

**lemma** *big-oh-poly-const*: *big-oh-poly*  $(\lambda n. c)$   
 ⟨proof⟩

**lemma** *big-oh-poly-poly*: *big-oh-poly*  $(\lambda n. n \wedge d)$   
 ⟨proof⟩

**lemma** *big-oh-poly-id*: *big-oh-poly*  $(\lambda n. n)$   
 ⟨proof⟩

**lemma** *big-oh-poly-le*:  
**assumes** *big-oh-poly*  $f$  **and**  $\bigwedge n. g\ n \leq f\ n$   
**shows** *big-oh-poly*  $g$   
 ⟨proof⟩

**lemma** *big-oh-poly-sum*:  
**assumes** *big-oh-poly*  $f1$  **and** *big-oh-poly*  $f2$   
**shows** *big-oh-poly*  $(\lambda n. f1\ n + f2\ n)$   
 ⟨proof⟩

**lemma** *big-oh-poly-prod*:  
**assumes** *big-oh-poly*  $f1$  **and** *big-oh-poly*  $f2$   
**shows** *big-oh-poly*  $(\lambda n. f1\ n * f2\ n)$   
 ⟨proof⟩

**lemma** *big-oh-poly-offset*:  
**assumes** *big-oh-poly*  $f$   
**shows**  $\exists b\ c\ d. d > 0 \wedge (\forall n. f\ n \leq b + c * n \wedge d)$   
 ⟨proof⟩

**lemma** *big-oh-poly-composition*:  
**assumes** *big-oh-poly*  $f1$  **and** *big-oh-poly*  $f2$   
**shows** *big-oh-poly*  $(f2 \circ f1)$   
 ⟨proof⟩

**lemma** *big-oh-poly-pow*:  
**fixes**  $f :: nat \Rightarrow nat$  **and**  $d :: nat$   
**assumes** *big-oh-poly*  $f$   
**shows** *big-oh-poly*  $(\lambda n. f\ n \wedge d)$   
 ⟨proof⟩

The textbook does not give an explicit definition of polynomials. It treats them as functions between natural numbers. So assuming the coefficients are natural numbers too, seems natural. We justify this choice when defining  $\mathcal{NP}$  in Section 3.1.

**definition** *polynomial* :: (nat  $\Rightarrow$  nat)  $\Rightarrow$  bool **where**  
*polynomial* f  $\equiv \exists cs. \forall n. f n = (\sum_{i \leftarrow [0..<length\ cs]}. cs ! i * n \wedge i)$

**lemma** *const-polynomial*: *polynomial* ( $\lambda\cdot c$ )  
 <proof>

**lemma** *polynomial-id*: *polynomial* id  
 <proof>

**lemma** *big-oh-poly-polynomial*:  
**fixes** f :: nat  $\Rightarrow$  nat  
**assumes** *polynomial* f  
**shows** *big-oh-poly* f  
 <proof>

## 2.2 Increasing the alphabet or the number of tapes

For technical reasons it is sometimes necessary to add tapes to a machine or to formally enlarge its alphabet such that it matches another machine's tape number or alphabet size without changing the behavior of the machine. The primary use of this is when composing machines with unequal alphabets or tape numbers (see Section 2.6).

### 2.2.1 Enlarging the alphabet

A Turing machine over alphabet  $G$  is not necessarily a Turing machine over a larger alphabet  $G' > G$  because reading a symbol in  $\{G, \dots, G' - 1\}$  the TM may write a symbol  $\geq G'$ . This is easy to remedy by modifying the TM to do nothing when it reads a symbol  $\geq G$ . It then formally satisfies the alphabet restriction property of Turing commands. This is rather crude, because the new TM loops infinitely on encountering a “forbidden” symbol, but it is good enough for our purposes.

The next function performs this transformation on a TM  $M$  over alphabet  $G$ . The resulting machine is a Turing machine for every alphabet size  $G' \geq G$ .

**definition** *enlarged* :: nat  $\Rightarrow$  machine  $\Rightarrow$  machine **where**  
*enlarged* G M  $\equiv \text{map } (\lambda \text{cmd } rs. \text{if symbols-lt } G \text{ } rs \text{ then cmd } rs \text{ else } (0, \text{map } (\lambda r. (r, \text{Stay})) rs)) M$

**lemma** *length-enlarged*: *length* (*enlarged* G M) = *length* M  
 <proof>

**lemma** *enlarged-nth*:  
**assumes** *symbols-lt* G gs **and**  $i < \text{length } M$   
**shows**  $(M ! i) gs = (\text{enlarged } G M ! i) gs$   
 <proof>

**lemma** *enlarged-write*:  
**assumes**  $\text{length } gs = k$  **and**  $i < \text{length } M$  **and** *turing-machine* k G M  
**shows**  $\text{length } (\text{snd } ((M ! i) gs)) = \text{length } (\text{snd } ((\text{enlarged } G M ! i) gs))$   
 <proof>

**lemma** *turing-machine-enlarged*:  
**assumes** *turing-machine* k G M **and**  $G' \geq G$   
**shows** *turing-machine* k G' (*enlarged* G M)  
 <proof>

The enlarged machine has the same behavior as the original machine when started on symbols over the original alphabet  $G$ .

**lemma** *execute-enlarged*:

**assumes** *turing-machine*  $k\ G\ M$  **and** *symbols-lt*  $G\ zs$   
**shows** *execute* (*enlarged*  $G\ M$ ) (*start-config*  $k\ zs$ )  $t = \text{execute } M\ (\text{start-config } k\ zs)\ t$   
⟨*proof*⟩

**lemma** *transforms-enlarged*:  
**assumes** *turing-machine*  $k\ G\ M$   
**and** *symbols-lt*  $G\ zs$   
**and** *transforms*  $M\ (\text{snd } (\text{start-config } k\ zs))\ t\ tps1$   
**shows** *transforms* (*enlarged*  $G\ M$ ) (*snd* (*start-config*  $k\ zs$ ))  $t\ tps1$   
⟨*proof*⟩

## 2.2.2 Increasing the number of tapes

We can add tapes to a Turing machine in such a way that on the additional tapes the machine does nothing. While the new tapes could go anywhere, we only consider appending them at the end or inserting them at the beginning.

### Appending tapes at the end

The next function turns a  $k$ -tape Turing machine into a  $k'$ -tape Turing machine (for  $k' \geq k$ ) by appending  $k' - k$  tapes at the end.

**definition** *append-tapes* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{machine} \Rightarrow \text{machine}$  **where**  
*append-tapes*  $k\ k'\ M \equiv$   
 $\text{map } (\lambda \text{cmd } rs. (\text{fst } (\text{cmd } (\text{take } k\ rs)), \text{snd } (\text{cmd } (\text{take } k\ rs)) @ (\text{map } (\lambda i. (rs ! i, \text{Stay})) [k..<k'])))\ M$

**lemma** *length-append-tapes*:  $\text{length } (\text{append-tapes } k\ k'\ M) = \text{length } M$   
⟨*proof*⟩

**lemma** *append-tapes-nth*:  
**assumes**  $i < \text{length } M$  **and**  $\text{length } gs = k'$   
**shows**  $(\text{append-tapes } k\ k'\ M ! i)\ gs =$   
 $(\text{fst } ((M ! i)\ (\text{take } k\ gs)), \text{snd } ((M ! i)\ (\text{take } k\ gs)) @ (\text{map } (\lambda j. (gs ! j, \text{Stay})) [k..<k']))$   
⟨*proof*⟩

**lemma** *append-tapes-tm*:  
**assumes** *turing-machine*  $k\ G\ M$  **and**  $k' \geq k$   
**shows** *turing-machine*  $k'\ G\ (\text{append-tapes } k\ k'\ M)$   
⟨*proof*⟩

**lemma** *execute-append-tapes*:  
**assumes** *turing-machine*  $k\ G\ M$  **and**  $k' \geq k$  **and**  $\text{length } tps = k'$   
**shows** *execute* (*append-tapes*  $k\ k'\ M$ ) ( $q, tps$ )  $t =$   
 $(\text{fst } (\text{execute } M\ (q, \text{take } k\ tps)\ t), \text{snd } (\text{execute } M\ (q, \text{take } k\ tps)\ t) @ \text{drop } k\ tps)$   
⟨*proof*⟩

**lemma** *execute-append-tapes'*:  
**assumes** *turing-machine*  $k\ G\ M$  **and**  $\text{length } tps = k$   
**shows** *execute* (*append-tapes*  $k\ (k + \text{length } tps')\ M$ ) ( $q, tps @ tps'$ )  $t =$   
 $(\text{fst } (\text{execute } M\ (q, tps)\ t), \text{snd } (\text{execute } M\ (q, tps)\ t) @ tps')$   
⟨*proof*⟩

**lemma** *transforms-append-tapes*:  
**assumes** *turing-machine*  $k\ G\ M$   
**and**  $\text{length } tps0 = k$   
**and** *transforms*  $M\ tps0\ t\ tps1$   
**shows** *transforms* (*append-tapes*  $k\ (k + \text{length } tps')\ M$ ) ( $tps0 @ tps'$ )  $t\ (tps1 @ tps')$   
**(is** *transforms* ? $M$  - - -)  
⟨*proof*⟩

## Inserting tapes at the beginning

The next function turns a  $k$ -tape Turing machine into a  $(k + d)$ -tape Turing machine by inserting  $d$  tapes at the beginning.

**definition** *prepend-tapes* ::  $\text{nat} \Rightarrow \text{machine} \Rightarrow \text{machine}$  **where**

*prepend-tapes*  $d$   $M \equiv$   
 $\text{map } (\lambda \text{cmd } rs. (\text{fst } (\text{cmd } (\text{drop } d \text{ rs})), \text{map } (\lambda h. (h, \text{Stay})) (\text{take } d \text{ rs}) @ \text{snd } (\text{cmd } (\text{drop } d \text{ rs})))) M$

**lemma** *prepend-tapes-at*:

**assumes**  $i < \text{length } M$

**shows**  $(\text{prepend-tapes } d M ! i) \text{ gs} =$

$(\text{fst } ((M ! i) (\text{drop } d \text{ gs})), \text{map } (\lambda h. (h, \text{Stay})) (\text{take } d \text{ gs}) @ \text{snd } ((M ! i) (\text{drop } d \text{ gs})))$

$\langle \text{proof} \rangle$

**lemma** *prepend-tapes-tm*:

**assumes** *turing-machine*  $k$   $G$   $M$

**shows** *turing-machine*  $(d + k)$   $G$   $(\text{prepend-tapes } d M)$

$\langle \text{proof} \rangle$

**definition** *shift-cfg* ::  $\text{tape list} \Rightarrow \text{config} \Rightarrow \text{config}$  **where**

*shift-cfg*  $tps$   $cfg \equiv (\text{fst } \text{cfg}, tps @ \text{snd } \text{cfg})$

**lemma** *execute-prepend-tapes*:

**assumes** *turing-machine*  $k$   $G$   $M$  **and**  $\text{length } tps = d$  **and**  $\|\text{cfg0}\| = k$

**shows** *execute*  $(\text{prepend-tapes } d M) (\text{shift-cfg } tps \text{ cfg0}) t = \text{shift-cfg } tps (\text{execute } M \text{ cfg0 } t)$

$\langle \text{proof} \rangle$

**lemma** *transforms-prepend-tapes*:

**assumes** *turing-machine*  $k$   $G$   $M$

**and**  $\text{length } tps = d$

**and**  $\text{length } tps0 = k$

**and** *transforms*  $M$   $tps0$   $t$   $tps1$

**shows** *transforms*  $(\text{prepend-tapes } d M) (tps @ tps0) t (tps @ tps1)$

$\langle \text{proof} \rangle$

**end**

## 2.3 Combining Turing machines

**theory** *Combinations*

**imports** *Basics HOL-Eisbach.Eisbach*

**begin**

This section describes how we can combine Turing machines in the way of traditional control structures found in structured programming, namely sequences, branching, and iterating. This allows us to build complex Turing machines from simpler ones and analyze their behavior and running time. Thanks to some syntactic sugar the result may even look like a programming language, but it is really more like a “construction kit” than a “true” programming language with small and big step semantics or Hoare rules. Instead we will merely have some lemmas for proving *transforms* statements for the combined machines. The remaining sections of this chapter will provide us with concrete Turing machines to combine.

### 2.3.1 Relocated machines

If we use a Turing machine  $M$  as part of another TM and there are  $q$  commands before  $M$ , then  $M$ 's target states will be off by  $q$ . This can be fixed by adding  $q$  to all target states of all commands in  $M$ , an operation we call *relocation*.

**definition** *relocate-cmd* ::  $\text{nat} \Rightarrow \text{command} \Rightarrow \text{command}$  **where**

*relocate-cmd*  $q$   $\text{cmd } rs \equiv (\text{fst } (\text{cmd } rs) + q, \text{snd } (\text{cmd } rs))$

**lemma** *relocate-cmd-head*:  $\text{relocate-cmd } q \text{ cmd } rs [\sim] j = \text{cmd } rs [\sim] j$

$\langle \text{proof} \rangle$

**lemma** *sem-relocate-cmd*:  $\text{sem } (\text{relocate-cmd } q \text{ cmd}) \text{ cfg} = (\text{sem } \text{cmd } \text{cfg}) <+==> q$   
 ⟨proof⟩

**definition** *relocate* ::  $\text{nat} \Rightarrow \text{machine} \Rightarrow \text{machine}$  **where**  
 $\text{relocate } q \text{ } M \equiv \text{map } (\text{relocate-cmd } q) \text{ } M$

**lemma** *relocate*:  
**assumes**  $M' = \text{relocate } q \text{ } M$  **and**  $i < \text{length } M$   
**shows**  $(M' ! i) \text{ } r = (\text{fst } ((M ! i) \text{ } r) + q, \text{snd } ((M ! i) \text{ } r))$   
 ⟨proof⟩

**lemma** *sem-relocate*:  
**assumes**  $M' = \text{relocate } q \text{ } M$  **and**  $i < \text{length } M$   
**shows**  $\text{sem } (M' ! i) \text{ } \text{cfg} = \text{sem } (M ! i) \text{ } \text{cfg} <+==> q$   
 ⟨proof⟩

**lemma** *turing-command-relocate-cmd*:  
**assumes** *turing-command*  $k \text{ } Q \text{ } G \text{ } \text{cmd}$   
**shows** *turing-command*  $k \text{ } (Q + q) \text{ } G \text{ } (\text{relocate-cmd } q \text{ } \text{cmd})$   
 ⟨proof⟩

**lemma** *turing-command-relocate*:  
**assumes**  $M' = \text{relocate } q \text{ } M$  **and** *turing-machine*  $k \text{ } G \text{ } M$  **and**  $i < \text{length } M$   
**shows** *turing-command*  $k \text{ } (\text{length } M + q) \text{ } G \text{ } (M' ! i)$   
 ⟨proof⟩

**lemma** *length-relocate*:  $\text{length } (\text{relocate } q \text{ } M) = \text{length } M$   
 ⟨proof⟩

**lemma** *relocate-jump-targets*:  
**assumes** *turing-machine*  $k \text{ } G \text{ } M$   
**and**  $M' = \text{relocate } q \text{ } M$   
**and**  $i < \text{length } M$   
**and**  $\text{length } rs = k$   
**shows**  $\text{fst } ((M' ! i) \text{ } rs) \leq \text{length } M + q$   
 ⟨proof⟩

**lemma** *relocate-zero*:  $\text{relocate } 0 \text{ } M = M$   
 ⟨proof⟩

## 2.3.2 Sequences

To execute two Turing machines sequentially we concatenate the two machines, relocating the second one by the length of the first one. In this way the halting state of the first machine becomes the start state of the second machine.

**definition** *turing-machine-sequential* ::  $\text{machine} \Rightarrow \text{machine} \Rightarrow \text{machine}$  (**infixl**  $<;>$  55) **where**  
 $M1 ;; M2 \equiv M1 @ (\text{relocate } (\text{length } M1) \text{ } M2)$

If the number of tapes and the alphabet size match, the concatenation of two Turing machines is again a Turing machine.

**lemma** *turing-machine-sequential-turing-machine* [*intro*, *simp*]:  
**assumes** *turing-machine*  $k \text{ } G \text{ } M1$  **and** *turing-machine*  $k \text{ } G \text{ } M2$   
**shows** *turing-machine*  $k \text{ } G \text{ } (M1 ;; M2)$  (**is** *turing-machine*  $k \text{ } G \text{ } ?M$ )  
 ⟨proof⟩

**lemma** *turing-machine-sequential-empty*: *turing-machine-sequential*  $[] \text{ } M = M$   
 ⟨proof⟩

**lemma** *turing-machine-sequential-nth*:  
**assumes**  $M = M1 ;; M2$  **and**  $p < \text{length } M2$   
**shows**  $M ! (p + \text{length } M1) = \text{relocate-cmd } (\text{length } M1) \text{ } (M2 ! p)$

*<proof>*

**lemma** *turing-machine-sequential-nth'*:

**assumes**  $M = M1 \ ;\ ;\ M2$  **and**  $\text{length } M1 \leq q$  **and**  $q < \text{length } M$   
**shows**  $M \ !\ q = \text{relocate-cmd } (\text{length } M1) (M2 \ !\ (q - \text{length } M1))$   
*<proof>*

**lemma** *length-turing-machine-sequential*:

$\text{length } (\text{turing-machine-sequential } M1 \ M2) = \text{length } M1 + \text{length } M2$   
*<proof>*

**lemma** *exe-relocate*:

$\text{exe } (M1 \ ;\ ;\ M2) (\text{cfg } <+==> \text{length } M1) = (\text{exe } M2 \ \text{cfg}) <+==> \text{length } M1$   
*<proof>*

**lemma** *execute-pre-append*:

**assumes**  $\text{halts } M1 \ \text{cfg}$  **and**  $\text{fst } \text{cfg} = 0$  **and**  $t \leq \text{running-time } M1 \ \text{cfg}$   
**shows**  $\text{execute } ((M0 \ ;\ ;\ M1) \ @ \ M2) (\text{cfg } <+==> \text{length } M0) \ t = (\text{execute } M1 \ \text{cfg } t) <+==> \text{length } M0$   
*<proof>*

**lemma** *transits-pre-append'*:

**assumes**  $\text{transforms } M1 \ \text{tps } t \ \text{tps}'$   
**shows**  $\text{transits } ((M0 \ ;\ ;\ M1) \ @ \ M2) (\text{length } M0, \ \text{tps}) \ t (\text{length } M0 + \text{length } M1, \ \text{tps}')$   
*<proof>*

**corollary** *transits-prepend*:

**assumes**  $\text{transforms } M1 \ \text{tps } t \ \text{tps}'$   
**shows**  $\text{transits } (M0 \ ;\ ;\ M1) (\text{length } M0, \ \text{tps}) \ t (\text{length } M0 + \text{length } M1, \ \text{tps}')$   
*<proof>*

**corollary** *transits-append*:

**assumes**  $\text{transforms } M1 \ \text{tps } t \ \text{tps}'$   
**shows**  $\text{transits } (M1 \ @ \ M2) (0, \ \text{tps}) \ t (\text{length } M1, \ \text{tps}')$   
*<proof>*

**corollary** *execute-append*:

**assumes**  $\text{fst } \text{cfg} = 0$  **and**  $\text{halts } M1 \ \text{cfg}$  **and**  $t \leq \text{running-time } M1 \ \text{cfg}$   
**shows**  $\text{execute } (M1 \ @ \ M2) \ \text{cfg } t = \text{execute } M1 \ \text{cfg } t$   
*<proof>*

**lemma** *execute-sequential*:

**assumes**  $\text{execute } M1 \ \text{cfg1 } t1 = \text{cfg1}'$   
**and**  $\text{fst } \text{cfg1} = 0$   
**and**  $t1 = \text{running-time } M1 \ \text{cfg1}$   
**and**  $\text{execute } M2 \ \text{cfg2 } t2 = \text{cfg2}'$   
**and**  $\text{cfg1}' = \text{cfg2 } <+==> \text{length } M1$   
**and**  $\text{halts } M1 \ \text{cfg1}$   
**shows**  $\text{execute } (M1 \ ;\ ;\ M2) \ \text{cfg1 } (t1 + t2) = \text{cfg2}' <+==> \text{length } M1$   
*<proof>*

The semantics and running time of the  $;$  operator:

**lemma** *transforms-turing-machine-sequential*:

**assumes**  $\text{transforms } M1 \ \text{tps1 } t1 \ \text{tps2}$  **and**  $\text{transforms } M2 \ \text{tps2 } t2 \ \text{tps3}$   
**shows**  $\text{transforms } (M1 \ ;\ ;\ M2) \ \text{tps1 } (t1 + t2) \ \text{tps3}$   
*<proof>*

**corollary** *transforms-tm-sequentialI*:

**assumes**  $\text{transforms } M1 \ \text{tps1 } t1 \ \text{tps2}$  **and**  $\text{transforms } M2 \ \text{tps2 } t2 \ \text{tps3}$  **and**  $t12 = t1 + t2$   
**shows**  $\text{transforms } (M1 \ ;\ ;\ M2) \ \text{tps1 } t12 \ \text{tps3}$   
*<proof>*

### 2.3.3 Branches

A branching Turing machine consists of a condition and two Turing machines, one for each of the branches. The condition can be any predicate over the list of symbols read from the tapes. The branching TM thus needs to perform conditional jumps, for which we will have the following command:

**definition** *cmd-jump* :: (symbol list  $\Rightarrow$  bool)  $\Rightarrow$  state  $\Rightarrow$  state  $\Rightarrow$  command **where**  
*cmd-jump* cond q1 q2 rs  $\equiv$  (if cond rs then q1 else q2, map ( $\lambda r$ . (r, Stay)) rs)

**lemma** *turing-command-jump-1*:  
**assumes**  $q1 \leq q2$  **and**  $k > 0$   
**shows** *turing-command* k q2 G (*cmd-jump* cond q1 q2)  
 <proof>

**lemma** *turing-command-jump-2*:  
**assumes**  $q2 \leq q1$  **and**  $k > 0$   
**shows** *turing-command* k q1 G (*cmd-jump* cond q1 q2)  
 <proof>

**lemma** *sem-jump-snd*: *snd* (sem (*cmd-jump* cond q1 q2) cfg) = *snd* cfg  
 <proof>

**lemma** *sem-jump-fst1*:  
**assumes** cond (read (snd cfg))  
**shows** *fst* (sem (*cmd-jump* cond q1 q2) cfg) = q1  
 <proof>

**lemma** *sem-jump-fst2*:  
**assumes**  $\neg$  cond (read (snd cfg))  
**shows** *fst* (sem (*cmd-jump* cond q1 q2) cfg) = q2  
 <proof>

**lemma** *sem-jump*:  
 sem (*cmd-jump* cond q1 q2) cfg = (if cond (config-read cfg) then q1 else q2, snd cfg)  
 <proof>

**lemma** *transits-jump*:  
 transits [*cmd-jump* cond q1 q2] (0, tps) 1 (if cond (read tps) then q1 else q2, tps)  
 <proof>

**definition** *turing-machine-branch* :: (symbol list  $\Rightarrow$  bool)  $\Rightarrow$  machine  $\Rightarrow$  machine  $\Rightarrow$  machine  
 (<IF - THEN - ELSE - ENDIF> 60)

**where**  
 IF cond THEN M1 ELSE M2 ENDIF  $\equiv$   
 [*cmd-jump* cond 1 (length M1 + 2)] @  
 (relocate 1 M1) @  
 [*cmd-jump* ( $\lambda$ -. True) (length M1 + length M2 + 2) 0] @  
 (relocate (length M1 + 2) M2)

**lemma** *turing-machine-branch-len*:  
 length (IF cond THEN M1 ELSE M2 ENDIF) = length M1 + length M2 + 2  
 <proof>

If the Turing machines for both branches have the same number of tapes and the same alphabet size, the branching machine is a Turing machine, too.

**lemma** *turing-machine-branch-turing-machine*:  
**assumes** *turing-machine* k G M1 **and** *turing-machine* k G M2  
**shows** *turing-machine* k G (IF cond THEN M1 ELSE M2 ENDIF)  
 (is *turing-machine* - - ?M)  
 <proof>

If the condition is true, the branching TM executes  $M_1$  and requires two extra steps: one for evaluating the condition and one for the unconditional jump to the halting state.

**lemma** *transforms-branch-true*:

**assumes** *transforms*  $M1$  *tps*  $t$  *tps'* **and** *cond* (*read tps*)  
**shows** *transforms* (*IF cond THEN M1 ELSE M2 ENDIF*) *tps* ( $t + 2$ ) *tps'*  
(is *transforms* ? $M$  - - )  
⟨*proof*⟩

If the condition is false, the branching TM executes  $M_2$  and requires one extra step to evaluate the condition.

**lemma** *transforms-branch-false*:  
**assumes** *transforms*  $M2$  *tps*  $t$  *tps'* **and**  $\neg$  *cond* (*read tps*)  
**shows** *transforms* (*IF cond THEN M1 ELSE M2 ENDIF*) *tps* ( $t + 1$ ) *tps'*  
(is *transforms* ? $M$  - - )  
⟨*proof*⟩

The behavior and running time of the branching Turing machine:

**lemma** *transforms-branch-full*:  
**assumes**  $c \implies$  *transforms*  $M1$  *tps*  $tT$  *tpsT*  
**and**  $\neg c \implies$  *transforms*  $M2$  *tps*  $tF$  *tpsF*  
**and**  $c \implies tT + 2 \leq t$   
**and**  $\neg c \implies tF + 1 \leq t$   
**and**  $c = \text{cond}$  (*read tps*)  
**and**  $tps' = (\text{if } c \text{ then } tpsT \text{ else } tpsF)$   
**shows** *transforms* (*IF cond THEN M1 ELSE M2 ENDIF*) *tps*  $t$  *tps'*  
⟨*proof*⟩

**corollary** *transforms-branchI*:  
**assumes** *cond* (*read tps*)  $\implies$  *transforms*  $M1$  *tps*  $tT$  *tpsT*  
**and**  $\neg$  *cond* (*read tps*)  $\implies$  *transforms*  $M2$  *tps*  $tF$  *tpsF*  
**and** *cond* (*read tps*)  $\implies tT + 2 \leq t$   
**and**  $\neg$  *cond* (*read tps*)  $\implies tF + 1 \leq t$   
**and** *cond* (*read tps*)  $\implies tps' = tpsT$   
**and**  $\neg$  *cond* (*read tps*)  $\implies tps' = tpsF$   
**shows** *transforms* (*IF cond THEN M1 ELSE M2 ENDIF*) *tps*  $t$  *tps'*  
⟨*proof*⟩

### 2.3.4 Loops

The loops are while loops consisting of a head and a body. The body is a Turing machine that is executed in every iteration as long as the condition in the head of the loop evaluates to true. The condition is of the same form as in branching TMs, namely a predicate over the symbols read from the tapes. Sometimes this is not expressive enough, and so we allow a Turing machine as part of the loop head that is run prior to evaluating the condition. In most cases, however, this TM is empty.

**definition** *turing-machine-loop* :: *machine*  $\Rightarrow$  (*symbol list*  $\Rightarrow$  *bool*)  $\Rightarrow$  *machine*  $\Rightarrow$  *machine*  
⟨*WHILE* - ; - *DO* - *DONE*⟩ 60

**where**

*WHILE*  $M1$  ; *cond* *DO*  $M2$  *DONE*  $\equiv$   
 $M1$  @  
[*cmd-jump* *cond* (*length*  $M1 + 1$ ) (*length*  $M1 + \text{length } M2 + 2$ )] @  
(*relocate* (*length*  $M1 + 1$ )  $M2$ ) @  
[*cmd-jump* ( $\lambda$ -. *True*) 0 0]

Intuitively the Turing machine *WHILE*  $M1$  ; *cond* *DO*  $M2$  *DONE* first executes  $M1$  and then checks the condition *cond*. If it is true, it executes  $M2$  and jumps back to the start state; otherwise it jumps to the halting state.

**lemma** *turing-machine-loop-len*:  
*length* (*WHILE*  $M1$  ; *cond* *DO*  $M2$  *DONE*) = *length*  $M1 + \text{length } M2 + 2$   
⟨*proof*⟩

If both Turing machines have the same number of tapes and alphabet size, then so does the looping Turing machine.

**lemma** *turing-machine-loop-turing-machine*:  
**assumes** *turing-machine*  $k$   $G$   $M1$  **and** *turing-machine*  $k$   $G$   $M2$

**shows** *turing-machine*  $k$   $G$  (*WHILE*  $M1$  ; *cond DO*  $M2$  *DONE*)  
 (**is** *turing-machine* - - ? $M$ )  
 ⟨*proof*⟩

**lemma** *transits-turing-machine-loop-cond-false*:  
**assumes** *transforms*  $M1$   $tps$   $t1$   $tps'$  **and**  $\neg$  *cond* (*read*  $tps'$ )  
**shows** *transits*  
 (*WHILE*  $M1$  ; *cond DO*  $M2$  *DONE*)  
 ( $0$ ,  $tps$ )  
 ( $t1 + 1$ )  
 (*length*  $M1$  + *length*  $M2$  +  $2$ ,  $tps'$ )  
 (**is** *transits* ? $M$  - - -)  
 ⟨*proof*⟩

**lemma** *transits-turing-machine-loop-cond-true*:  
**assumes** *transforms*  $M1$   $tps$   $t1$   $tps'$   
**and** *transforms*  $M2$   $tps'$   $t2$   $tps''$   
**and** *cond* (*read*  $tps'$ )  
**shows** *transits*  
 (*WHILE*  $M1$  ; *cond DO*  $M2$  *DONE*)  
 ( $0$ ,  $tps$ )  
 ( $t1 + t2 + 2$ )  
 ( $0$ ,  $tps''$ )  
 (**is** *transits* ? $M$  - - -)  
 ⟨*proof*⟩

In this article we will only encounter while loops that are in fact for loops, that is, where the number of iterations is known beforehand. Moreover, using the same time bound for every iteration will lead to a good enough upper bound for the entire loop.

The *transforms* rule for a loop with  $m$  iterations where the running time of both TMs is bounded by a constant:

**lemma** *transforms-loop-for*:  
**fixes**  $m$   $t1$   $t2$  :: *nat*  
**and**  $M1$   $M2$  :: *machine*  
**and**  $tps$  :: *nat*  $\Rightarrow$  *tape list*  
**and**  $tps'$  :: *nat*  $\Rightarrow$  *tape list*  
**assumes**  $\bigwedge i. i \leq m \Rightarrow$  *transforms*  $M1$  ( $tps$   $i$ )  $t1$  ( $tps'$   $i$ )  
**assumes**  $\bigwedge i. i < m \Rightarrow$  *transforms*  $M2$  ( $tps'$   $i$ )  $t2$  ( $tps$  (*Suc*  $i$ ))  
**and**  $\bigwedge i. i < m \Rightarrow$  *cond* (*read* ( $tps'$   $i$ ))  
**and**  $\neg$  *cond* (*read* ( $tps'$   $m$ ))  
**assumes**  $t1t \geq m * (t1 + t2 + 2) + t1 + 1$   
**shows** *transforms*  
 (*WHILE*  $M1$  ; *cond DO*  $M2$  *DONE*)  
 ( $tps$   $0$ )  
 $t1t$   
 ( $tps'$   $m$ )  
 ⟨*proof*⟩

The rule becomes even simpler in the common case in which the Turing machine in the loop head is empty.

**lemma** *transforms-loop-simple*:  
**fixes**  $m$   $t$  :: *nat*  
**and**  $M$  :: *machine*  
**and**  $tps$  :: *nat*  $\Rightarrow$  *tape list*  
**assumes**  $\bigwedge i. i < m \Rightarrow$  *transforms*  $M$  ( $tps$   $i$ )  $t$  ( $tps$  (*Suc*  $i$ ))  
**and**  $\bigwedge i. i < m \Rightarrow$  *cond* (*read* ( $tps$   $i$ ))  
**and**  $\neg$  *cond* (*read* ( $tps$   $m$ ))  
**assumes**  $t1t \geq m * (t + 2) + 1$   
**shows** *transforms*  
 (*WHILE* [] ; *cond DO*  $M$  *DONE*)  
 ( $tps$   $0$ )  
 $t1t$   
 ( $tps$   $m$ )

*<proof>*

### 2.3.5 A proof method

Statements about the behavior and running time of Turing machines, expressed via the predicate *transforms*, are the most common ones we are going to prove. The following proof method applies introduction rules for this predicate. These rules are either the ones we proved for the control structures (sequence, branching, loop) or the ones describing the semantics of concrete Turing machines. The rules of the second kind are collected in the attribute *transforms-intros*.

Applying such a rule usually leaves three kinds of goals: some simple ones requiring only instantiation of schematic variables; one for the equality of two tape lists; and one for the time bound. For the last two goals the proof method offers parameters *tps* and *time*, respectively.

I have to admit that most of the details of the proof method were determined by trial and error.

```
named-theorems transforms-intros
declare transforms-Nil [transforms-intros]

method tform uses tps time declares transforms-intros =
(
  ((rule transforms-tm-sequentialI) +
   | rule transforms-branchI
   | rule transforms-loop-simple
   | rule transforms-loop-for
   | rule transforms-intros)
  ; (rule transforms-intros)?
  ; (simp only; fail)?
  ; ((use tps in simp; fail) | (use time in simp; fail))?
  ; (match conclusion in left = right for left right :: tape list
     ⇒ <(fastforce simp add: tps list-update-swap; fail)>)?
  ; (match conclusion in left = right for left right :: nat
     ⇒ <(use time in simp; fail)?>)?
)
```

These lemmas are sometimes helpful for proving the equality of tape lists:

```
lemma list-update-swap-less:  $i' < i \implies ys[i := x, i' := x'] = ys[i' := x', i := x]$ 
<proof>
```

```
lemma nth-list-update-neq':  $j \neq i \implies xs[i := x] ! j = xs ! j$ 
<proof>
```

**end**

## 2.4 Elementary Turing machines

```
theory Elementary
  imports Combinations
begin
```

In this section we devise some simple yet useful Turing machines. We have already fully analyzed the empty TM, where start and halting state coincide, in the lemmas *computes-Nil-empty*, *Nil-tm*, and *transforms-Nil*. The next more complex TMs are those with exactly one command. They represent TMs with two states: the halting state and the start state where the action happens. This action might last for one step only, or the TM may stay in this state for longer; for example, it can move a tape head rightward to the next blank symbol. We will also start using the *;* operator to combine some of the one-command TMs.

Most Turing machines we are going to construct throughout this section and indeed the entire article are really families of Turing machines that usually are parameterized by tape indices.

```
type-synonym tapeidx = nat
```

Throughout this article, names of commands are prefixed with *cmd-* and names of Turing machines with *tm-*. Usually for a TM named *tm-foo* there is a lemma *tm-foo-tm* stating that it really is a Turing machine and a lemma *transforms-tm-fooI* describing its semantics and running time. The lemma usually receives a *transforms-intros* attribute for use with our proof method.

If *tm-foo* comprises more than two TMs we will typically analyze the semantics and running time in a locale named *turing-machine-foo*. The first example of this is *tm-equals* in Section 2.4.10.

When it comes to running times, we will have almost no scruples simplifying upper bounds to have the form  $a + b \cdot n^c$  for some constants  $a, b, c$ , even if this means, for example, bounding  $n \log n$  by  $n^2$ .

### 2.4.1 Clean tapes

Most of our Turing machines will not change the start symbol in the first cell of a tape nor will they write the start symbol anywhere else. The only exceptions are machines that simulate arbitrary other machines. We call tapes that have the start symbol only in the first cell *clean tapes*.

**definition** *clean-tape* :: *tape*  $\Rightarrow$  *bool* **where**

*clean-tape* *tp*  $\equiv \forall i. \text{fst } tp \ i = \triangleright \longleftrightarrow i = 0$

**lemma** *clean-tapeI*:

**assumes**  $\bigwedge i. \text{fst } tp \ i = \triangleright \longleftrightarrow i = 0$

**shows** *clean-tape* *tp*

*<proof>*

**lemma** *clean-tapeI'*:

**assumes** *fst* *tp*  $0 = \triangleright$  **and**  $\bigwedge i. i > 0 \implies \text{fst } tp \ i \neq \triangleright$

**shows** *clean-tape* *tp*

*<proof>*

A clean configuration is one with only clean tapes.

**definition** *clean-config* :: *config*  $\Rightarrow$  *bool* **where**

*clean-config* *cfg*  $\equiv (\forall j < |cfg|. \forall i. (cfg \ <:> \ j) \ i = \triangleright \longleftrightarrow i = 0)$

**lemma** *clean-config-tapes*: *clean-config* *cfg* =  $(\forall tp \in \text{set } (snd \ cfg). \text{clean-tape } tp)$

*<proof>*

**lemma** *clean-configI*:

**assumes**  $\bigwedge j \ i. j < \text{length } tps \implies \text{fst } (tps \ ! \ j) \ i = \triangleright \longleftrightarrow i = 0$

**shows** *clean-config* (*q*, *tps*)

*<proof>*

**lemma** *clean-configI'*:

**assumes**  $\bigwedge tp \ i. tp \in \text{set } tps \implies \text{fst } tp \ i = \triangleright \longleftrightarrow i = 0$

**shows** *clean-config* (*q*, *tps*)

*<proof>*

**lemma** *clean-configI''*:

**assumes**  $\bigwedge tp. tp \in \text{set } tps \implies (\text{fst } tp \ 0 = \triangleright \wedge (\forall i > 0. \text{fst } tp \ i \neq \triangleright))$

**shows** *clean-config* (*q*, *tps*)

*<proof>*

**lemma** *clean-configE*:

**assumes** *clean-config* (*q*, *tps*)

**shows**  $\bigwedge j \ i. j < \text{length } tps \implies \text{fst } (tps \ ! \ j) \ i = \triangleright \longleftrightarrow i = 0$

*<proof>*

**lemma** *clean-configE'*:

**assumes** *clean-config* (*q*, *tps*)

**shows**  $\bigwedge tp \ i. tp \in \text{set } tps \implies \text{fst } tp \ i = \triangleright \longleftrightarrow i = 0$

*<proof>*

**lemma** *clean-contents-proper* [*simp*]: *proper-symbols* *zs*  $\implies \text{clean-tape } ([zs], q)$

*<proof>*

**lemma** *contents-clean-tape'*:  $\text{proper-symbols } zs \implies \text{fst } tp = \lfloor zs \rfloor \implies \text{clean-tape } tp$   
 ⟨proof⟩

Some more lemmas about *contents*:

**lemma** *contents-append-blanks*:  $\lfloor ys @ \text{replicate } m \square \rfloor = \lfloor ys \rfloor$   
 ⟨proof⟩

**lemma** *contents-append-update*:  
**assumes**  $\text{length } ys = m$   
**shows**  $\lfloor ys @ [v] @ zs \rfloor (\text{Suc } m := w) = \lfloor ys @ [w] @ zs \rfloor$   
 ⟨proof⟩

**lemma** *contents-snoc*:  $\lfloor ys \rfloor (\text{Suc } (\text{length } ys) := w) = \lfloor ys @ [w] \rfloor$   
 ⟨proof⟩

**definition** *config-update-pos* ::  $\text{config} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{config}$  **where**  
 $\text{config-update-pos } cfg \ j \ p \equiv (\text{fst } cfg, (\text{snd } cfg)[j := (\text{cfg} \langle : \rangle j, p)])$

**lemma** *config-update-pos-0*:  $\text{config-update-pos } cfg \ j \ (\text{cfg} \langle \# \rangle j) = \text{cfg}$   
 ⟨proof⟩

**definition** *config-update-fwd* ::  $\text{config} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{config}$  **where**  
 $\text{config-update-fwd } cfg \ j \ d \equiv (\text{fst } cfg, (\text{snd } cfg)[j := (\text{cfg} \langle : \rangle j, \text{cfg} \langle \# \rangle j + d)])$

**lemma** *config-update-fwd-0*:  $\text{config-update-fwd } cfg \ j \ 0 = \text{cfg}$   
 ⟨proof⟩

**lemma** *config-update-fwd-additive*:  
 $\text{config-update-fwd } (\text{config-update-fwd } cfg \ j \ d1) \ j \ d2 = (\text{config-update-fwd } cfg \ j \ (d1 + d2))$   
 ⟨proof⟩

## 2.4.2 Moving tape heads

Among the most simple things a Turing machine can do is moving one of its tape heads.

### Moving left

The next command makes a TM move its head on tape  $j$  one cell to the left unless, of course, it is in the leftmost cell already.

**definition** *cmd-left* ::  $\text{tapeidx} \Rightarrow \text{command}$  **where**  
 $\text{cmd-left } j \equiv \lambda rs. (1, \text{map } (\lambda i. (rs \ ! \ i, \text{if } i = j \text{ then Left else Stay})) [0..<\text{length } rs])$

**lemma** *turing-command-left*:  $\text{turing-command } k \ 1 \ G \ (\text{cmd-left } j)$   
 ⟨proof⟩

**lemma** *cmd-left'*:  $[*] \ (\text{cmd-left } j \ rs) = 1$   
 ⟨proof⟩

**lemma** *cmd-left''*:  $j < \text{length } rs \implies (\text{cmd-left } j \ rs) \ [!] \ j = (rs \ ! \ j, \text{Left})$   
 ⟨proof⟩

**lemma** *cmd-left'''*:  $i < \text{length } rs \implies i \neq j \implies (\text{cmd-left } j \ rs) \ [!] \ i = (rs \ ! \ i, \text{Stay})$   
 ⟨proof⟩

**lemma** *tape-list-eq*:  
**assumes**  $\text{length } tps' = \text{length } tps$   
**and**  $\bigwedge i. i < \text{length } tps \implies i \neq j \implies tps' \ ! \ i = tps \ ! \ i$   
**and**  $tps' \ ! \ j = x$   
**shows**  $tps' = tps[j := x]$   
 ⟨proof⟩

**lemma** *sem-cmd-left*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{sem } (\text{cmd-left } j) (0, tps) = (1, tps[j := (\text{fst } (tps ! j), \text{snd } (tps ! j) - 1)])$   
 $\langle \text{proof} \rangle$

**definition** *tm-left* :: *tapeidx*  $\Rightarrow$  *machine* **where**  
 $\text{tm-left } j \equiv [\text{cmd-left } j]$

**lemma** *tm-left-tm*:  $k \geq 2 \implies G \geq 4 \implies \text{turing-machine } k \ G \ (\text{tm-left } j)$   
 $\langle \text{proof} \rangle$

**lemma** *exe-tm-left*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{exe } (\text{tm-left } j) (0, tps) = (1, tps[j := tps ! j \mid - \mid 1])$   
 $\langle \text{proof} \rangle$

**lemma** *execute-tm-left*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{execute } (\text{tm-left } j) (0, tps) (\text{Suc } 0) = (1, tps[j := tps ! j \mid - \mid 1])$   
 $\langle \text{proof} \rangle$

**lemma** *transits-tm-left*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{transits } (\text{tm-left } j) (0, tps) (\text{Suc } 0) (1, tps[j := tps ! j \mid - \mid 1])$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-left*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{transforms } (\text{tm-left } j) \ tps \ (\text{Suc } 0) \ (tps[j := tps ! j \mid - \mid 1])$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-leftI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$   
**and**  $t = 1$   
**and**  $tps' = tps[j := tps ! j \mid - \mid 1]$   
**shows**  $\text{transforms } (\text{tm-left } j) \ tps \ t \ tps'$   
 $\langle \text{proof} \rangle$

## Moving right

The next command makes the head on tape  $j$  move one cell to the right.

**definition** *cmd-right* :: *tapeidx*  $\Rightarrow$  *command* **where**  
 $\text{cmd-right } j \equiv \lambda rs. (1, \text{map } (\lambda i. (rs ! i, \text{if } i = j \text{ then } \text{Right} \text{ else } \text{Stay})) [0..<\text{length } rs])$

**lemma** *turing-command-right*: *turing-command*  $k \ 1 \ G \ (\text{cmd-right } j)$   
 $\langle \text{proof} \rangle$

**lemma** *cmd-right'*:  $[\ast] \ (\text{cmd-right } j \ rs) = 1$   
 $\langle \text{proof} \rangle$

**lemma** *cmd-right''*:  $j < \text{length } rs \implies (\text{cmd-right } j \ rs) \ [!] \ j = (rs ! j, \text{Right})$   
 $\langle \text{proof} \rangle$

**lemma** *cmd-right'''*:  $i < \text{length } rs \implies i \neq j \implies (\text{cmd-right } j \ rs) \ [!] \ i = (rs ! i, \text{Stay})$   
 $\langle \text{proof} \rangle$

**lemma** *sem-cmd-right*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{sem } (\text{cmd-right } j) (0, tps) = (1, tps[j := (\text{fst } (tps ! j), \text{snd } (tps ! j) + 1)])$   
 $\langle \text{proof} \rangle$

**definition** *tm-right* :: *tapeidx*  $\Rightarrow$  *machine* **where**  
 $\text{tm-right } j \equiv [\text{cmd-right } j]$

**lemma** *tm-right-tm*:  $k \geq 2 \implies G \geq 4 \implies \text{turing-machine } k \ G \ (tm\text{-right } j)$   
 ⟨proof⟩

**lemma** *exe-tm-right*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{exe } (tm\text{-right } j) \ (0, tps) = (1, tps[j:=fst \ (tps \ ! \ j), \ snd \ (tps \ ! \ j) \ + \ 1])$   
 ⟨proof⟩

**lemma** *execute-tm-right*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{execute } (tm\text{-right } j) \ (0, tps) \ (Suc \ 0) = (1, tps[j:=fst \ (tps \ ! \ j), \ snd \ (tps \ ! \ j) \ + \ 1])$   
 ⟨proof⟩

**lemma** *transits-tm-right*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{transits } (tm\text{-right } j) \ (0, tps) \ (Suc \ 0) \ (1, tps[j:=fst \ (tps \ ! \ j), \ snd \ (tps \ ! \ j) \ + \ 1])$   
 ⟨proof⟩

**lemma** *transforms-tm-right*:  
**assumes**  $j < \text{length } tps$   
**shows**  $\text{transforms } (tm\text{-right } j) \ tps \ (Suc \ 0) \ (tps[j := tps \ ! \ j \ + \ 1])$   
 ⟨proof⟩

**lemma** *transforms-tm-rightI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$   
**and**  $t = Suc \ 0$   
**and**  $tps' = tps[j := tps \ ! \ j \ + \ 1]$   
**shows**  $\text{transforms } (tm\text{-right } j) \ tps \ t \ tps'$   
 ⟨proof⟩

## Moving right on several tapes

The next command makes the heads on all tapes from a set  $J$  of tapes move one cell to the right.

**definition** *cmd-right-many* :: *tapeidx set*  $\Rightarrow$  *command* **where**  
 $\text{cmd-right-many } J \equiv \lambda rs. \ (1, \text{map } (\lambda i. \ (rs \ ! \ i, \ \text{if } i \in J \ \text{then } Right \ \text{else } Stay))) \ [0..<\text{length } rs])$

**lemma** *turing-command-right-many*: *turing-command*  $k \ 1 \ G \ (\text{cmd-right-many } J)$   
 ⟨proof⟩

**lemma** *sem-cmd-right-many*:  
 $\text{sem } (\text{cmd-right-many } J) \ (0, tps) = (1, \text{map } (\lambda j. \ \text{if } j \in J \ \text{then } tps \ ! \ j \ + \ 1 \ \text{else } tps \ ! \ j) \ [0..<\text{length } tps])$   
 ⟨proof⟩

**definition** *tm-right-many* :: *tapeidx set*  $\Rightarrow$  *machine* **where**  
 $\text{tm-right-many } J \equiv [\text{cmd-right-many } J]$

**lemma** *tm-right-many-tm*:  $k \geq 2 \implies G \geq 4 \implies \text{turing-machine } k \ G \ (tm\text{-right-many } J)$   
 ⟨proof⟩

**lemma** *transforms-tm-right-manyI* [*transforms-intros*]:  
**assumes**  $t = Suc \ 0$   
**and**  $tps' = \text{map } (\lambda j. \ \text{if } j \in J \ \text{then } tps \ ! \ j \ + \ 1 \ \text{else } tps \ ! \ j) \ [0..<\text{length } tps]$   
**shows**  $\text{transforms } (tm\text{-right-many } J) \ tps \ t \ tps'$   
 ⟨proof⟩

### 2.4.3 Copying and translating tape contents

The Turing machines in this section scan a tape  $j_1$  and copy the symbols to another tape  $j_2$ . Scanning can be performed in either direction, and “copying” may include mapping the symbols.

## Copying and translating from one tape to another

The next predicate is true iff. on the given tape the next symbol from the set  $H$  of symbols is exactly  $n$  cells to the right from the current head position. Thus, a command that moves the tape head right until it finds a symbol from  $H$  takes  $n$  steps and moves the head  $n$  cells right.

**definition**  $rneigh :: \text{tape} \Rightarrow \text{symbol set} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
 $rneigh\ tp\ H\ n \equiv \text{fst}\ tp\ (\text{snd}\ tp + n) \in H \wedge (\forall n' < n. \text{fst}\ tp\ (\text{snd}\ tp + n') \notin H)$

**lemma**  $rneighI$ :  
**assumes**  $\text{fst}\ tp\ (\text{snd}\ tp + n) \in H$  **and**  $\bigwedge n'. n' < n \implies \text{fst}\ tp\ (\text{snd}\ tp + n') \notin H$   
**shows**  $rneigh\ tp\ H\ n$   
 $\langle \text{proof} \rangle$

The analogous predicate for moving to the left:

**definition**  $lneigh :: \text{tape} \Rightarrow \text{symbol set} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
 $lneigh\ tp\ H\ n \equiv \text{fst}\ tp\ (\text{snd}\ tp - n) \in H \wedge (\forall n' < n. \text{fst}\ tp\ (\text{snd}\ tp - n') \notin H)$

**lemma**  $lneighI$ :  
**assumes**  $\text{fst}\ tp\ (\text{snd}\ tp - n) \in H$  **and**  $\bigwedge n'. n' < n \implies \text{fst}\ tp\ (\text{snd}\ tp - n') \notin H$   
**shows**  $lneigh\ tp\ H\ n$   
 $\langle \text{proof} \rangle$

The next command scans tape  $j_1$  rightward until it reaches a symbol from the set  $H$ . While doing so it copies the symbols, after applying a mapping  $f$ , to tape  $j_2$ .

**definition**  $\text{cmd-trans-until} :: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{symbol set} \Rightarrow (\text{symbol} \Rightarrow \text{symbol}) \Rightarrow \text{command}$  **where**  
 $\text{cmd-trans-until}\ j_1\ j_2\ H\ f \equiv \lambda rs. \\
\text{if } rs ! j_1 \in H \\
\text{then } (1, \text{map } (\lambda r. (r, \text{Stay}))\ rs) \\
\text{else } (0, \text{map } (\lambda i. (\text{if } i = j_2 \text{ then } f\ (rs ! j_1) \text{ else } rs ! i, \text{if } i = j_1 \vee i = j_2 \text{ then } \text{Right} \text{ else } \text{Stay}))\ [0..<\text{length}\ rs])$

The analogous command for moving to the left:

**definition**  $\text{cmd-ltrans-until} :: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{symbol set} \Rightarrow (\text{symbol} \Rightarrow \text{symbol}) \Rightarrow \text{command}$  **where**  
 $\text{cmd-ltrans-until}\ j_1\ j_2\ H\ f \equiv \lambda rs. \\
\text{if } rs ! j_1 \in H \\
\text{then } (1, \text{map } (\lambda r. (r, \text{Stay}))\ rs) \\
\text{else } (0, \text{map } (\lambda i. (\text{if } i = j_2 \text{ then } f\ (rs ! j_1) \text{ else } rs ! i, \text{if } i = j_1 \vee i = j_2 \text{ then } \text{Left} \text{ else } \text{Stay}))\ [0..<\text{length}\ rs])$

**lemma**  $\text{proper-cmd-trans-until}$ :  $\text{proper-command}\ k\ (\text{cmd-trans-until}\ j_1\ j_2\ H\ f)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{proper-cmd-ltrans-until}$ :  $\text{proper-command}\ k\ (\text{cmd-ltrans-until}\ j_1\ j_2\ H\ f)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sem-cmd-trans-until-1}$ :  
**assumes**  $j_1 < k$  **and**  $\text{length}\ tps = k$  **and**  $(0, tps) <.> j_1 \in H$   
**shows**  $\text{sem}\ (\text{cmd-trans-until}\ j_1\ j_2\ H\ f)\ (0, tps) = (1, tps)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sem-cmd-ltrans-until-1}$ :  
**assumes**  $j_1 < k$  **and**  $\text{length}\ tps = k$  **and**  $(0, tps) <.> j_1 \in H$   
**shows**  $\text{sem}\ (\text{cmd-ltrans-until}\ j_1\ j_2\ H\ f)\ (0, tps) = (1, tps)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sem-cmd-trans-until-2}$ :  
**assumes**  $j_1 < k$  **and**  $\text{length}\ tps = k$  **and**  $(0, tps) <.> j_1 \notin H$   
**shows**  $\text{sem}\ (\text{cmd-trans-until}\ j_1\ j_2\ H\ f)\ (0, tps) = \\
(0, tps[j_1 := tps ! j_1 \mid + \mid 1, j_2 := tps ! j_2 \mid := \mid (f\ (tps :: j_1)) \mid + \mid 1])$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sem-cmd-ltrans-until-2}$ :  
**assumes**  $j_1 < k$  **and**  $\text{length}\ tps = k$  **and**  $(0, tps) <.> j_1 \notin H$   
**shows**  $\text{sem}\ (\text{cmd-ltrans-until}\ j_1\ j_2\ H\ f)\ (0, tps) =$

$(0, tps[j1 := tps ! j1 \mid - \mid 1, j2 := tps ! j2 \mid := \mid (f (tps :: j1)) \mid - \mid 1])$   
 ⟨proof⟩

**definition** *tm-trans-until* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *symbol set*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *machine* **where**  
*tm-trans-until* *j1 j2 H f*  $\equiv$  [*cmd-trans-until* *j1 j2 H f*]

**definition** *tm-ltrans-until* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *symbol set*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *machine* **where**  
*tm-ltrans-until* *j1 j2 H f*  $\equiv$  [*cmd-ltrans-until* *j1 j2 H f*]

**lemma** *tm-trans-until-tm*:

**assumes**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $\forall h < G. f h < G$  **and**  $k \geq 2$  **and**  $G \geq 4$   
**shows** *turing-machine* *k G (tm-trans-until j1 j2 H f)*  
 ⟨proof⟩

**lemma** *tm-ltrans-until-tm*:

**assumes**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $\forall h < G. f h < G$  **and**  $k \geq 2$  **and**  $G \geq 4$   
**shows** *turing-machine* *k G (tm-ltrans-until j1 j2 H f)*  
 ⟨proof⟩

**lemma** *exe-tm-trans-until-1*:

**assumes**  $j1 < k$  **and** *length tps = k* **and**  $(0, tps) <.> j1 \in H$   
**shows** *exe (tm-trans-until j1 j2 H f) (0, tps) = (1, tps)*  
 ⟨proof⟩

**lemma** *exe-tm-ltrans-until-1*:

**assumes**  $j1 < k$  **and** *length tps = k* **and**  $(0, tps) <.> j1 \in H$   
**shows** *exe (tm-ltrans-until j1 j2 H f) (0, tps) = (1, tps)*  
 ⟨proof⟩

**lemma** *exe-tm-trans-until-2*:

**assumes**  $j1 < k$  **and** *length tps = k* **and**  $(0, tps) <.> j1 \notin H$   
**shows** *exe (tm-trans-until j1 j2 H f) (0, tps) =*  
 $(0, tps[j1 := tps ! j1 \mid + \mid 1, j2 := tps ! j2 \mid := \mid (f (tps :: j1)) \mid + \mid 1])$   
 ⟨proof⟩

**lemma** *exe-tm-ltrans-until-2*:

**assumes**  $j1 < k$  **and** *length tps = k* **and**  $(0, tps) <.> j1 \notin H$   
**shows** *exe (tm-ltrans-until j1 j2 H f) (0, tps) =*  
 $(0, tps[j1 := tps ! j1 \mid - \mid 1, j2 := tps ! j2 \mid := \mid (f (tps :: j1)) \mid - \mid 1])$   
 ⟨proof⟩

Let *tp<sub>1</sub>* and *tp<sub>2</sub>* be two tapes with head positions *i<sub>1</sub>* and *i<sub>2</sub>*, respectively. The next function describes the result of overwriting the symbols at positions *i<sub>2</sub>, ..., i<sub>2</sub> + n - 1* on tape *tp<sub>2</sub>* by the symbols at positions *i<sub>1</sub>, ..., i<sub>1</sub> + n - 1* on tape *tp<sub>1</sub>* after applying a symbol map *f*.

**definition** *transplant* :: *tape*  $\Rightarrow$  *tape*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *nat*  $\Rightarrow$  *tape* **where**

*transplant* *tp1 tp2 f n*  $\equiv$   
 $(\lambda i. \text{if } \text{snd } tp2 \leq i \wedge i < \text{snd } tp2 + n \text{ then } f (\text{fst } tp1 (\text{snd } tp1 + i - \text{snd } tp2)) \text{ else } \text{fst } tp2 \ i,$   
 $\text{snd } tp2 + n)$

The analogous function for moving to the left while copying:

**definition** *ltransplant* :: *tape*  $\Rightarrow$  *tape*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *nat*  $\Rightarrow$  *tape* **where**

*ltransplant* *tp1 tp2 f n*  $\equiv$   
 $(\lambda i. \text{if } \text{snd } tp2 - n < i \wedge i \leq \text{snd } tp2 \text{ then } f (\text{fst } tp1 (\text{snd } tp1 + i - \text{snd } tp2)) \text{ else } \text{fst } tp2 \ i,$   
 $\text{snd } tp2 - n)$

**lemma** *transplant-0*: *transplant* *tp1 tp2 f 0 = tp2*

⟨proof⟩

**lemma** *ltransplant-0*: *ltransplant* *tp1 tp2 f 0 = tp2*

⟨proof⟩

**lemma** *transplant-upd*: *transplant* *tp1 tp2 f n*  $\mid := \mid (f ( \mid (tp1 \mid + \mid n))) \mid + \mid 1 = \text{transplant } tp1 \ tp2 \ f \ (Suc \ n)$

⟨proof⟩

**lemma** *ltransplant-upd*:

**assumes**  $n < \text{snd } tp2$

**shows**  $\text{ltransplant } tp1 \ tp2 \ f \ n \ |:=| \ (f \ (| \cdot| \ (tp1 \ |-\ | \ n))) \ |-\ | \ 1 = \text{ltransplant } tp1 \ tp2 \ f \ (\text{Suc } n)$

*<proof>*

**lemma** *tapes-ltransplant-upd*:

**assumes**  $t < \text{tps} \ :\# \ j2$  **and**  $t < \text{tps} \ :\# \ j1$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } \text{tps} = k$

**and**  $\text{tps}' = \text{tps}[j1 := \text{tps} ! j1 \ |-\ | \ t, j2 := \text{ltransplant } (\text{tps} ! j1) \ (\text{tps} ! j2) \ f \ t]$

**shows**  $\text{tps}'[j1 := \text{tps}' ! j1 \ |-\ | \ 1, j2 := \text{tps}' ! j2 \ |:=| \ (f \ (\text{tps}' \ :: \ j1)) \ |-\ | \ 1] =$

$\text{tps}[j1 := \text{tps} ! j1 \ |-\ | \ \text{Suc } t, j2 := \text{ltransplant } (\text{tps} ! j1) \ (\text{tps} ! j2) \ f \ (\text{Suc } t)]$

**(is ?lhs = ?rhs)**

*<proof>*

**lemma** *execute-tm-trans-until-less*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } \text{tps} = k$  **and**  $\text{rneigh } (\text{tps} ! j1) \ H \ n$  **and**  $t \leq n$

**shows**  $\text{execute } (\text{tm-trans-until } j1 \ j2 \ H \ f) \ (0, \ \text{tps}) \ t =$

$(0, \ \text{tps}[j1 := \text{tps} ! j1 \ |+\ | \ t, j2 := \text{transplant } (\text{tps} ! j1) \ (\text{tps} ! j2) \ f \ t])$

*<proof>*

**lemma** *execute-tm-ltrans-until-less*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } \text{tps} = k$

**and**  $\text{lneigh } (\text{tps} ! j1) \ H \ n$

**and**  $t \leq n$

**and**  $n \leq \text{tps} \ :\# \ j1$

**and**  $n \leq \text{tps} \ :\# \ j2$

**shows**  $\text{execute } (\text{tm-ltrans-until } j1 \ j2 \ H \ f) \ (0, \ \text{tps}) \ t =$

$(0, \ \text{tps}[j1 := \text{tps} ! j1 \ |-\ | \ t, j2 := \text{ltransplant } (\text{tps} ! j1) \ (\text{tps} ! j2) \ f \ t])$

*<proof>*

**lemma** *execute-tm-trans-until*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } \text{tps} = k$  **and**  $\text{rneigh } (\text{tps} ! j1) \ H \ n$

**shows**  $\text{execute } (\text{tm-trans-until } j1 \ j2 \ H \ f) \ (0, \ \text{tps}) \ (\text{Suc } n) =$

$(1, \ \text{tps}[j1 := \text{tps} ! j1 \ |+\ | \ n, j2 := \text{transplant } (\text{tps} ! j1) \ (\text{tps} ! j2) \ f \ n])$

*<proof>*

**lemma** *execute-tm-ltrans-until*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } \text{tps} = k$

**and**  $\text{lneigh } (\text{tps} ! j1) \ H \ n$

**and**  $n \leq \text{tps} \ :\# \ j1$

**and**  $n \leq \text{tps} \ :\# \ j2$

**shows**  $\text{execute } (\text{tm-ltrans-until } j1 \ j2 \ H \ f) \ (0, \ \text{tps}) \ (\text{Suc } n) =$

$(1, \ \text{tps}[j1 := \text{tps} ! j1 \ |-\ | \ n, j2 := \text{ltransplant } (\text{tps} ! j1) \ (\text{tps} ! j2) \ f \ n])$

*<proof>*

**lemma** *transits-tm-trans-until*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } \text{tps} = k$  **and**  $\text{rneigh } (\text{tps} ! j1) \ H \ n$

**shows**  $\text{transits } (\text{tm-trans-until } j1 \ j2 \ H \ f)$

$(0, \ \text{tps})$

$(\text{Suc } n)$

$(1, \ \text{tps}[j1 := \text{tps} ! j1 \ |+\ | \ n, j2 := \text{transplant } (\text{tps} ! j1) \ (\text{tps} ! j2) \ f \ n])$

*<proof>*

**lemma** *transits-tm-ltrans-until*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } \text{tps} = k$

**and**  $\text{lneigh } (\text{tps} ! j1) \ H \ n$

**and**  $n \leq \text{tps} \ :\# \ j1$

**and**  $n \leq \text{tps} \ :\# \ j2$

**shows**  $\text{transits } (\text{tm-ltrans-until } j1 \ j2 \ H \ f)$

$(0, \ \text{tps})$

$(\text{Suc } n)$

$(1, \ \text{tps}[j1 := \text{tps} ! j1 \ |-\ | \ n, j2 := \text{ltransplant } (\text{tps} ! j1) \ (\text{tps} ! j2) \ f \ n])$

*<proof>*

**lemma** *transforms-tm-trans-until*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$  **and**  $\text{rneigh } (tps ! j1) H n$   
**shows**  $\text{transforms } (tm\text{-trans-until } j1 j2 H f)$   
 $tps$   
 $(Suc\ n)$   
 $(tps[j1 := tps ! j1 \mid+] n, j2 := \text{transplant } (tps ! j1) (tps ! j2) f n)$   
 $\langle proof \rangle$

**lemma** *transforms-tm-ltrans-until*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{lneigh } (tps ! j1) H n$   
**and**  $n \leq tps : \# : j1$   
**and**  $n \leq tps : \# : j2$   
**shows**  $\text{transforms } (tm\text{-ltrans-until } j1 j2 H f)$   
 $tps$   
 $(Suc\ n)$   
 $(tps[j1 := tps ! j1 \mid-] n, j2 := \text{ltransplant } (tps ! j1) (tps ! j2) f n)$   
 $\langle proof \rangle$

**corollary** *transforms-tm-trans-untilI* [*transforms-intros*]:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{rneigh } (tps ! j1) H n$   
**and**  $t = Suc\ n$   
**and**  $tps' = tps[j1 := tps ! j1 \mid+] n, j2 := \text{transplant } (tps ! j1) (tps ! j2) f n$   
**shows**  $\text{transforms } (tm\text{-trans-until } j1 j2 H f) tps\ t\ tps'$   
 $\langle proof \rangle$

**corollary** *transforms-tm-ltrans-untilI* [*transforms-intros*]:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{lneigh } (tps ! j1) H n$   
**and**  $n \leq tps : \# : j1$   
**and**  $n \leq tps : \# : j2$   
**and**  $t = Suc\ n$   
**and**  $tps' = tps[j1 := tps ! j1 \mid-] n, j2 := \text{ltransplant } (tps ! j1) (tps ! j2) f n$   
**shows**  $\text{transforms } (tm\text{-ltrans-until } j1 j2 H f) tps\ t\ tps'$   
 $\langle proof \rangle$

## Copying one tape to another

If we set the symbol map  $f$  in *tm-trans-until* to the identity function, we get a Turing machine that simply makes a copy.

**definition** *tm-cp-until* ::  $\text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{symbol set} \Rightarrow \text{machine}$  **where**  
 $tm\text{-cp-until } j1 j2 H \equiv tm\text{-trans-until } j1 j2 H id$

**lemma** *id-symbol*:  $\forall h < G. (id :: \text{symbol} \Rightarrow \text{symbol}) h < G$   
 $\langle proof \rangle$

**lemma** *tm-cp-until-tm*:

**assumes**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $G \geq 4$   
**shows**  $\text{turing-machine } k\ G\ (tm\text{-cp-until } j1 j2 H)$   
 $\langle proof \rangle$

**abbreviation** *implant* ::  $\text{tape} \Rightarrow \text{tape} \Rightarrow \text{nat} \Rightarrow \text{tape}$  **where**

$implant\ tp1\ tp2\ n \equiv \text{transplant } tp1\ tp2\ id\ n$

**lemma** *implant*:  $implant\ tp1\ tp2\ n =$

$(\lambda i. \text{if } snd\ tp2 \leq i \wedge i < snd\ tp2 + n \text{ then } fst\ tp1\ (snd\ tp1 + i - snd\ tp2) \text{ else } fst\ tp2\ i,$   
 $snd\ tp2 + n)$

$\langle proof \rangle$

**lemma** *implantI*:

**assumes**  $tp' =$

( $\lambda i$ . if  $\text{snd } tp2 \leq i \wedge i < \text{snd } tp2 + n$  then  $\text{fst } tp1 (\text{snd } tp1 + i - \text{snd } tp2)$  else  $\text{fst } tp2 i$ ,  
 $\text{snd } tp2 + n$ )

**shows**  $\text{implant } tp1 \ tp2 \ n = tp'$

$\langle \text{proof} \rangle$

**lemma** *fst-snd-pair*:  $\text{fst } t = a \implies \text{snd } t = b \implies t = (a, b)$

$\langle \text{proof} \rangle$

**lemma** *implantI'*:

**assumes**  $\text{fst } tp' =$

( $\lambda i$ . if  $\text{snd } tp2 \leq i \wedge i < \text{snd } tp2 + n$  then  $\text{fst } tp1 (\text{snd } tp1 + i - \text{snd } tp2)$  else  $\text{fst } tp2 i$ )

**and**  $\text{snd } tp' = \text{snd } tp2 + n$

**shows**  $\text{implant } tp1 \ tp2 \ n = tp'$

$\langle \text{proof} \rangle$

**lemma** *implantI''*:

**assumes**  $\bigwedge i$ .  $\text{snd } tp2 \leq i \wedge i < \text{snd } tp2 + n \implies \text{fst } tp' i = \text{fst } tp1 (\text{snd } tp1 + i - \text{snd } tp2)$

**and**  $\bigwedge i$ .  $i < \text{snd } tp2 \implies \text{fst } tp' i = \text{fst } tp2 i$

**and**  $\bigwedge i$ .  $\text{snd } tp2 + n \leq i \implies \text{fst } tp' i = \text{fst } tp2 i$

**assumes**  $\text{snd } tp' = \text{snd } tp2 + n$

**shows**  $\text{implant } tp1 \ tp2 \ n = tp'$

$\langle \text{proof} \rangle$

**lemma** *implantI'''*:

**assumes**  $\bigwedge i$ .  $i2 \leq i \wedge i < i2 + n \implies \text{ys } i = \text{ys1 } (i1 + i - i2)$

**and**  $\bigwedge i$ .  $i < i2 \implies \text{ys } i = \text{ys2 } i$

**and**  $\bigwedge i$ .  $i2 + n \leq i \implies \text{ys } i = \text{ys2 } i$

**assumes**  $i = i2 + n$

**shows**  $\text{implant } (\text{ys1}, i1) (\text{ys2}, i2) \ n = (\text{ys}, i)$

$\langle \text{proof} \rangle$

**lemma** *implant-self*:  $\text{implant } tp \ tp \ n = tp \ |+\ | \ n$

$\langle \text{proof} \rangle$

**lemma** *transforms-tm-cp-untill* [*transforms-intros*]:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$

**and**  $\text{rneigh } (tps \ ! \ j1) \ H \ n$

**and**  $t = \text{Suc } n$

**and**  $tps' = tps[j1 := tps \ ! \ j1 \ |+\ | \ n, j2 := \text{implant } (tps \ ! \ j1) (tps \ ! \ j2) \ n]$

**shows**  $\text{transforms } (\text{tm-cp-until } j1 \ j2 \ H) \ tps \ t \ tps'$

$\langle \text{proof} \rangle$

**lemma** *implant-contents*:

**assumes**  $i > 0$  **and**  $n + (i - 1) \leq \text{length } xs$

**shows**  $\text{implant } ([xs], i) ([ys], \text{Suc } (\text{length } ys)) \ n =$

$([ys \ @ \ (\text{take } n \ (\text{drop } (i - 1) \ xs))], \text{Suc } (\text{length } ys) + n)$

(**is**  $?lhs = ?rhs$ )

$\langle \text{proof} \rangle$

## Moving to the next specific symbol

Copying a tape to itself means just moving to the right.

**definition** *tm-right-until* ::  $\text{tapeidx} \Rightarrow \text{symbol set} \Rightarrow \text{machine}$  **where**

$\text{tm-right-until } j \ H \equiv \text{tm-cp-until } j \ j \ H$

Copying a tape to itself does not change the tape. So this is a Turing machine even for the input tape  $j = 0$ , unlike *tm-cp-until* where the target tape cannot, in general, be the input tape.

**lemma** *tm-right-until-tm*:

**assumes**  $j < k$  **and**  $k \geq 2$  **and**  $G \geq 4$

**shows**  $\text{turing-machine } k \ G \ (\text{tm-right-until } j \ H)$

$\langle \text{proof} \rangle$

**lemma** *transforms-tm-right-untill* [*transforms-intros*]:

**assumes**  $j < \text{length } tps$   
**and**  $\text{rneigh } (tps ! j) H n$   
**and**  $t = \text{Suc } n$   
**and**  $tps' = (tps[j := tps ! j | + | n])$   
**shows**  $\text{transforms } (tm\text{-right-until } j H) tps t tps'$   
 $\langle \text{proof} \rangle$

## Translating to a constant symbol

Another way to specialize  $tm\text{-trans-until}$  and  $tm\text{-ltrans-until}$  is to have a constant function  $f$ .

**definition**  $tm\text{-const-until} :: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{symbol set} \Rightarrow \text{symbol} \Rightarrow \text{machine}$  **where**  
 $tm\text{-const-until } j1 j2 H h \equiv tm\text{-trans-until } j1 j2 H (\lambda-. h)$

**lemma**  $tm\text{-const-until-tm}$ :  
**assumes**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $h < G$  **and**  $k \geq 2$  **and**  $G \geq 4$   
**shows**  $\text{turing-machine } k G (tm\text{-const-until } j1 j2 H h)$   
 $\langle \text{proof} \rangle$

Continuing with our fantasy names ending in  $\text{-plant}$ , we name the operation  $\text{constplant}$ .

**abbreviation**  $\text{constplant} :: \text{tape} \Rightarrow \text{symbol} \Rightarrow \text{nat} \Rightarrow \text{tape}$  **where**  
 $\text{constplant } tp2 h n \equiv \text{transplant } (\lambda-. 0, 0) tp2 (\lambda-. h) n$

**lemma**  $\text{constplant-transplant}$ :  $\text{constplant } tp2 h n = \text{transplant } tp1 tp2 (\lambda-. h) n$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{constplant}$ :  $\text{constplant } tp2 h n =$   
 $(\lambda i. \text{if } \text{snd } tp2 \leq i \wedge i < \text{snd } tp2 + n \text{ then } h \text{ else } \text{fst } tp2 i,$   
 $\text{snd } tp2 + n)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{transforms-tm-const-untilI}$  [ $\text{transforms-intros}$ ]:  
**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{rneigh } (tps ! j1) H n$   
**and**  $t = \text{Suc } n$   
**and**  $tps' = tps[j1 := tps ! j1 | + | n, j2 := \text{constplant } (tps ! j2) h n]$   
**shows**  $\text{transforms } (tm\text{-const-until } j1 j2 H h) tps t tps'$   
 $\langle \text{proof} \rangle$

**definition**  $tm\text{-lconst-until} :: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{symbol set} \Rightarrow \text{symbol} \Rightarrow \text{machine}$  **where**  
 $tm\text{-lconst-until } j1 j2 H h \equiv tm\text{-ltrans-until } j1 j2 H (\lambda-. h)$

**lemma**  $tm\text{-lconst-until-tm}$ :  
**assumes**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $h < G$  **and**  $k \geq 2$  **and**  $G \geq 4$   
**shows**  $\text{turing-machine } k G (tm\text{-lconst-until } j1 j2 H h)$   
 $\langle \text{proof} \rangle$

**abbreviation**  $\text{lconstplant} :: \text{tape} \Rightarrow \text{symbol} \Rightarrow \text{nat} \Rightarrow \text{tape}$  **where**  
 $\text{lconstplant } tp2 h n \equiv \text{ltransplant } (\lambda-. 0, 0) tp2 (\lambda-. h) n$

**lemma**  $\text{lconstplant-ltransplant}$ :  $\text{lconstplant } tp2 h n = \text{ltransplant } tp1 tp2 (\lambda-. h) n$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{lconstplant}$ :  $\text{lconstplant } tp2 h n =$   
 $(\lambda i. \text{if } \text{snd } tp2 - n < i \wedge i \leq \text{snd } tp2 \text{ then } h \text{ else } \text{fst } tp2 i,$   
 $\text{snd } tp2 - n)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{transforms-tm-lconst-untilI}$  [ $\text{transforms-intros}$ ]:  
**assumes**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{lneigh } (tps ! j1) H n$   
**and**  $n \leq tps \text{ :\#} j1$   
**and**  $n \leq tps \text{ :\#} j2$   
**and**  $t = \text{Suc } n$

**and**  $tps' = tps[j1 := tps ! j1 \mid n, j2 := lconstplant (tps ! j2) h n]$   
**shows** *transforms (tm-lconst-until j1 j2 H h) tps t tps'*  
 ⟨*proof*⟩

## 2.4.4 Writing single symbols

The next command writes a fixed symbol  $h$  to tape  $j$ . It does not move a tape head.

**definition** *cmd-write* :: *tapeidx*  $\Rightarrow$  *symbol*  $\Rightarrow$  *command* **where**  
 $cmd\text{-}write\ j\ h\ rs \equiv (1, \text{map } (\lambda i. (if\ i = j\ then\ h\ else\ rs\ !\ i, Stay))\ [0..<length\ rs])$

**lemma** *sem-cmd-write*:  $sem\ (cmd\text{-}write\ j\ h)\ (0, tps) = (1, tps[j := tps ! j \mid h])$   
 ⟨*proof*⟩

**definition** *tm-write* :: *tapeidx*  $\Rightarrow$  *symbol*  $\Rightarrow$  *machine* **where**  
 $tm\text{-}write\ j\ h \equiv [cmd\text{-}write\ j\ h]$

**lemma** *tm-write-tm*:  
**assumes**  $0 < j$  **and**  $j < k$  **and**  $h < G$  **and**  $G \geq 4$   
**shows** *turing-machine*  $k\ G\ (tm\text{-}write\ j\ h)$   
 ⟨*proof*⟩

**lemma** *transforms-tm-writeI* [*transforms-intros*]:  
**assumes**  $tps' = tps[j := tps ! j \mid h]$   
**shows** *transforms (tm-write j h) tps 1 tps'*  
 ⟨*proof*⟩

The next command writes the symbol to tape  $j_2$  that results from applying a function  $f$  to the symbol read from tape  $j_1$ . It does not move any tape heads.

**definition** *cmd-trans2* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *command* **where**  
 $cmd\text{-}trans2\ j1\ j2\ f\ rs \equiv (1, \text{map } (\lambda i. (if\ i = j2\ then\ f\ (rs\ !\ j1)\ else\ rs\ !\ i, Stay))\ [0..<length\ rs])$

**lemma** *sem-cmd-trans2*:  
**assumes**  $j1 < length\ tps$   
**shows**  $sem\ (cmd\text{-}trans2\ j1\ j2\ f)\ (0, tps) = (1, tps[j2 := tps ! j2 \mid :=\ (f\ (tps\ ::\ j1))])$   
 ⟨*proof*⟩

**definition** *tm-trans2* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *machine* **where**  
 $tm\text{-}trans2\ j1\ j2\ f \equiv [cmd\text{-}trans2\ j1\ j2\ f]$

**lemma** *tm-trans2-tm*:  
**assumes**  $j1 < k$  **and**  $0 < j2$  **and**  $j2 < k$  **and**  $\forall h < G. f\ h < G$  **and**  $k \geq 2$  **and**  $G \geq 4$   
**shows** *turing-machine*  $k\ G\ (tm\text{-}trans2\ j1\ j2\ f)$   
 ⟨*proof*⟩

**lemma** *exe-tm-trans2*:  
**assumes**  $j1 < length\ tps$   
**shows**  $exe\ (tm\text{-}trans2\ j1\ j2\ f)\ (0, tps) = (1, tps[j2 := tps ! j2 \mid :=\ (f\ (tps\ ::\ j1))])$   
 ⟨*proof*⟩

**lemma** *execute-tm-trans2*:  
**assumes**  $j1 < length\ tps$   
**shows**  $execute\ (tm\text{-}trans2\ j1\ j2\ f)\ (0, tps)\ 1 = (1, tps[j2 := tps ! j2 \mid :=\ (f\ (tps\ ::\ j1))])$   
 ⟨*proof*⟩

**lemma** *transits-tm-trans2*:  
**assumes**  $j1 < length\ tps$   
**shows**  $transits\ (tm\text{-}trans2\ j1\ j2\ f)\ (0, tps)\ 1\ (1, tps[j2 := tps ! j2 \mid :=\ (f\ (tps\ ::\ j1))])$   
 ⟨*proof*⟩

**lemma** *transforms-tm-trans2*:  
**assumes**  $j1 < length\ tps$   
**shows**  $transforms\ (tm\text{-}trans2\ j1\ j2\ f)\ tps\ 1\ (tps[j2 := tps ! j2 \mid :=\ (f\ (tps\ ::\ j1))])$   
 ⟨*proof*⟩

**lemma** *transforms-tm-trans2I* [*transforms-intros*]:  
**assumes**  $j1 < \text{length } tps$  **and**  $tps' = tps[j2 := tps ! j2 |:=| (f (tps :: j1))]$   
**shows** *transforms* (*tm-trans2*  $j1 j2 f$ )  $tps$  1  $tps'$   
 $\langle \text{proof} \rangle$

Equating the two tapes in *tm-trans2*, we can map a symbol in-place.

**definition** *tm-trans* :: *tapeidx*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *machine* **where**  
*tm-trans*  $j f \equiv \text{tm-trans2 } j j f$

**lemma** *tm-trans-tm*:  
**assumes**  $0 < j$  **and**  $j < k$  **and**  $\forall h < G. f h < G$  **and**  $G \geq 4$   
**shows** *turing-machine*  $k G$  (*tm-trans*  $j f$ )  
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-transI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$  **and**  $tps' = tps[j := tps ! j |:=| (f (tps :: j))]$   
**shows** *transforms* (*tm-trans*  $j f$ )  $tps$  1  $tps'$   
 $\langle \text{proof} \rangle$

The next command is like the previous one, except it also moves the tape head to the right.

**definition** *cmd-rtrans* :: *tapeidx*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *command* **where**  
*cmd-rtrans*  $j f rs \equiv (1, \text{map } (\lambda i. (\text{if } i = j \text{ then } f (rs ! i) \text{ else } rs ! i, \text{if } i = j \text{ then Right else Stay})) [0..<\text{length } rs])$

**lemma** *sem-cmd-rtrans*:  
**assumes**  $j < \text{length } tps$   
**shows** *sem* (*cmd-rtrans*  $j f$ ) ( $0, tps$ ) = ( $1, tps[j := tps ! j |:=| (f (tps :: j)) | + 1]$ )  
 $\langle \text{proof} \rangle$

**definition** *tm-rtrans* :: *tapeidx*  $\Rightarrow$  (*symbol*  $\Rightarrow$  *symbol*)  $\Rightarrow$  *machine* **where**  
*tm-rtrans*  $j f \equiv [\text{cmd-rtrans } j f]$

**lemma** *tm-rtrans-tm*:  
**assumes**  $0 < j$  **and**  $j < k$  **and**  $\forall h < G. f h < G$  **and**  $k \geq 2$  **and**  $G \geq 4$   
**shows** *turing-machine*  $k G$  (*tm-rtrans*  $j f$ )  
 $\langle \text{proof} \rangle$

**lemma** *exe-tm-rtrans*:  
**assumes**  $j < \text{length } tps$   
**shows** *exe* (*tm-rtrans*  $j f$ ) ( $0, tps$ ) = ( $1, tps[j := tps ! j |:=| (f (tps :: j)) | + 1]$ )  
 $\langle \text{proof} \rangle$

**lemma** *execute-tm-rtrans*:  
**assumes**  $j < \text{length } tps$   
**shows** *execute* (*tm-rtrans*  $j f$ ) ( $0, tps$ ) 1 = ( $1, tps[j := tps ! j |:=| (f (tps :: j)) | + 1]$ )  
 $\langle \text{proof} \rangle$

**lemma** *transits-tm-rtrans*:  
**assumes**  $j < \text{length } tps$   
**shows** *transits* (*tm-rtrans*  $j f$ ) ( $0, tps$ ) 1 ( $1, tps[j := tps ! j |:=| (f (tps :: j)) | + 1]$ )  
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-rtrans*:  
**assumes**  $j < \text{length } tps$   
**shows** *transforms* (*tm-rtrans*  $j f$ )  $tps$  1 ( $tps[j := tps ! j |:=| (f (tps :: j)) | + 1]$ )  
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-rtransI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$  **and**  $tps' = tps[j := tps ! j |:=| (f (tps :: j)) | + 1]$   
**shows** *transforms* (*tm-rtrans*  $j f$ )  $tps$  1  $tps'$   
 $\langle \text{proof} \rangle$

The next command writes a fixed symbol  $h$  to all tapes in the set  $J$ .

**definition** *cmd-write-many* :: *tapeidx set*  $\Rightarrow$  *symbol*  $\Rightarrow$  *command* **where**  
*cmd-write-many* *J h rs*  $\equiv$  (*1*, *map* ( $\lambda i$ . (*if* *i*  $\in$  *J* *then* *h* *else* *rs* ! *i*, Stay)) [0..*length* *rs*])

**lemma** *proper-cmd-write-many*: *proper-command* *k* (*cmd-write-many* *J h*)  
 <proof>

**lemma** *sem-cmd-write-many*:  
**shows** *sem* (*cmd-write-many* *J h*) (*0*, *tps*) =  
 (*1*, *map* ( $\lambda j$ . *if* *j*  $\in$  *J* *then* *tps* ! *j*  $|\!:=|$  *h* *else* *tps* ! *j*) [0..*length* *tps*])  
 <proof>

**definition** *tm-write-many* :: *tapeidx set*  $\Rightarrow$  *symbol*  $\Rightarrow$  *machine* **where**  
*tm-write-many* *J h*  $\equiv$  [*cmd-write-many* *J h*]

**lemma** *tm-write-many-tm*:  
**assumes**  $0 \notin J$  **and**  $\forall j \in J. j < k$  **and**  $h < G$  **and**  $k \geq 2$  **and**  $G \geq 4$   
**shows** *turing-machine* *k G* (*tm-write-many* *J h*)  
 <proof>

**lemma** *exe-tm-write-many*: *exe* (*tm-write-many* *J h*) (*0*, *tps*) =  
 (*1*, *map* ( $\lambda j$ . *if* *j*  $\in$  *J* *then* *tps* ! *j*  $|\!:=|$  *h* *else* *tps* ! *j*) [0..*length* *tps*])  
 <proof>

**lemma** *execute-tm-write-many*: *execute* (*tm-write-many* *J h*) (*0*, *tps*) *1* =  
 (*1*, *map* ( $\lambda j$ . *if* *j*  $\in$  *J* *then* *tps* ! *j*  $|\!:=|$  *h* *else* *tps* ! *j*) [0..*length* *tps*])  
 <proof>

**lemma** *transforms-tm-write-many*:  
*transforms* (*tm-write-many* *J h*) *tps* *1* (*map* ( $\lambda j$ . *if* *j*  $\in$  *J* *then* *tps* ! *j*  $|\!:=|$  *h* *else* *tps* ! *j*) [0..*length* *tps*])  
 <proof>

**lemma** *transforms-tm-write-manyI* [*transforms-intros*]:  
**assumes**  $\forall j \in J. j < k$   
**and** *length* *tps* = *k*  
**and** *length* *tps'* = *k*  
**and**  $\bigwedge j. j \in J \Longrightarrow tps' ! j = tps ! j$   
**and**  $\bigwedge j. j < k \Longrightarrow j \notin J \Longrightarrow tps' ! j = tps ! j$   
**shows** *transforms* (*tm-write-many* *J h*) *tps* *1* *tps'*  
 <proof>

## 2.4.5 Writing a symbol multiple times

In this section we devise a Turing machine that writes the symbol sequence  $h^m$  with a hard-coded symbol *h* and number *m* to a tape. The resulting tape is defined by the next operation:

**definition** *overwrite* :: *tape*  $\Rightarrow$  *symbol*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape* **where**  
*overwrite* *tp h m*  $\equiv$  ( $\lambda i$ . *if* *snd* *tp*  $\leq i \wedge i < \text{snd } tp + m$  *then* *h* *else* *fst* *tp* *i*, *snd* *tp* + *m*)

**lemma** *overwrite-0*: *overwrite* *tp h* *0* = *tp*  
 <proof>

**lemma** *overwrite-upd*: (*overwrite* *tp h* *t*)  $|\!:=|$  *h*  $|\!+|$  *1* = *overwrite* *tp h* (*Suc* *t*)  
 <proof>

**lemma** *overwrite-upd'*:  
**assumes**  $j < \text{length } tps$  **and**  $tps' = tps[j := \text{overwrite } (tps ! j) h t]$   
**shows**  $(tps[j := \text{overwrite } (tps ! j) h t])[j := tps' ! j |\!:=| h |\!+| 1] =$   
 $tps[j := \text{overwrite } (tps ! j) h (\text{Suc } t)]$   
 <proof>

The next command writes the symbol *h* to the tape *j* and moves the tape head to the right.

**definition** *cmd-char* :: *tapeidx*  $\Rightarrow$  *symbol*  $\Rightarrow$  *command* **where**  
*cmd-char* *j z* = *cmd-rtrans* *j* ( $\lambda$ -. *z*)

**lemma** *turing-command-char*:  
**assumes**  $0 < j$  **and**  $j < k$  **and**  $h < G$   
**shows** *turing-command*  $k$   $1$   $G$  (*cmd-char*  $j$   $h$ )  
 $\langle$ *proof* $\rangle$

**definition** *tm-char* :: *tapeidx*  $\Rightarrow$  *symbol*  $\Rightarrow$  *machine* **where**  
*tm-char*  $j$   $z$   $\equiv$  [*cmd-char*  $j$   $z$ ]

**lemma** *tm-char-tm*:  
**assumes**  $0 < j$  **and**  $j < k$  **and**  $G \geq 4$  **and**  $z < G$   
**shows** *turing-machine*  $k$   $G$  (*tm-char*  $j$   $z$ )  
 $\langle$ *proof* $\rangle$

**lemma** *transforms-tm-charI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$  **and**  $tps' = tps[j := tps ! j |:=| z | + | 1]$   
**shows** *transforms* (*tm-char*  $j$   $z$ )  $tps$   $1$   $tps'$   
 $\langle$ *proof* $\rangle$

**lemma** *sem-cmd-char*:  
**assumes**  $j < \text{length } tps$   
**shows** *sem* (*cmd-char*  $j$   $h$ )  $(0, tps) = (1, tps[j := tps ! j |:=| h | + | 1])$   
 $\langle$ *proof* $\rangle$

The next TM is a sequence of  $m$  *cmd-char* commands properly relocated. It writes a sequence of  $m$  times the symbol  $h$  to tape  $j$ .

**definition** *tm-write-repeat* :: *tapeidx*  $\Rightarrow$  *symbol*  $\Rightarrow$  *nat*  $\Rightarrow$  *machine* **where**  
*tm-write-repeat*  $j$   $h$   $m$   $\equiv$  *map* ( $\lambda i. \text{relocate-cmd } i$  (*cmd-char*  $j$   $h$ ))  $[0..<m]$

**lemma** *tm-write-repeat-tm*:  
**assumes**  $0 < j$  **and**  $j < k$  **and**  $h < G$  **and**  $k \geq 2$  **and**  $G \geq 4$   
**shows** *turing-machine*  $k$   $G$  (*tm-write-repeat*  $j$   $h$   $m$ )  
 $\langle$ *proof* $\rangle$

**lemma** *exe-tm-write-repeat*:  
**assumes**  $j < \text{length } tps$  **and**  $q < m$   
**shows** *exe* (*tm-write-repeat*  $j$   $h$   $m$ )  $(q, tps) = (\text{Suc } q, tps[j := tps ! j |:=| h | + | 1])$   
 $\langle$ *proof* $\rangle$

**lemma** *execute-tm-write-repeat*:  
**assumes**  $j < \text{length } tps$  **and**  $t \leq m$   
**shows** *execute* (*tm-write-repeat*  $j$   $h$   $m$ )  $(0, tps) t = (t, tps[j := \text{overwrite } (tps ! j) h t])$   
 $\langle$ *proof* $\rangle$

**lemma** *transforms-tm-write-repeatI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$  **and**  $tps' = tps[j := \text{overwrite } (tps ! j) h m]$   
**shows** *transforms* (*tm-write-repeat*  $j$   $h$   $m$ )  $tps$   $m$   $tps'$   
 $\langle$ *proof* $\rangle$

## 2.4.6 Moving to the start of the tape

The next command moves the head on tape  $j$  to the left until it reaches a symbol from the set  $H$ :

**definition** *cmd-left-until* :: *symbol set*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *command* **where**  
*cmd-left-until*  $H$   $j$   $rs$   $\equiv$   
*if*  $rs ! j \in H$   
*then*  $(1, \text{map } (\lambda r. (r, \text{Stay})) rs)$   
*else*  $(0, \text{map } (\lambda i. (rs ! i, \text{if } i = j \text{ then Left else Stay})) [0..<\text{length } rs])$

**lemma** *sem-cmd-left-until-1*:  
**assumes**  $j < k$  **and**  $\text{length } tps = k$  **and**  $(0, tps) <.> j \in H$   
**shows** *sem* (*cmd-left-until*  $H$   $j$ )  $(0, tps) = (1, tps)$   
 $\langle$ *proof* $\rangle$

**lemma** *sem-cmd-left-until-2*:

**assumes**  $j < k$  **and**  $\text{length } tps = k$  **and**  $(0, tps) \langle . \rangle j \notin H$   
**shows**  $\text{sem } (\text{cmd-left-until } H \ j) \ (0, tps) = (0, tps[j := tps ! j \mid - \ 1])$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tm-left-until} :: \text{symbol set} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**  
 $\text{tm-left-until } H \ j \equiv [\text{cmd-left-until } H \ j]$

**lemma**  $\text{tm-left-until-tm}$ :  
**assumes**  $k \geq 2$  **and**  $G \geq 4$   
**shows**  $\text{turing-machine } k \ G \ (\text{tm-left-until } H \ j)$   
 $\langle \text{proof} \rangle$

A *begin tape* for a set of symbols has one of these symbols only in cell zero. It generalizes the concept of clean tapes, where the set of symbols is  $\{\triangleright\}$ .

**definition**  $\text{begin-tape} :: \text{symbol set} \Rightarrow \text{tape} \Rightarrow \text{bool}$  **where**  
 $\text{begin-tape } H \ tp \equiv \forall i. \text{fst } tp \ i \in H \longleftrightarrow i = 0$

**lemma**  $\text{begin-tapeI}$ :  
**assumes**  $\text{fst } tp \ 0 \in H$  **and**  $\bigwedge i. i > 0 \implies \text{fst } tp \ i \notin H$   
**shows**  $\text{begin-tape } H \ tp$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{exe-tm-left-until-1}$ :  
**assumes**  $j < \text{length } tps$  **and**  $(0, tps) \langle . \rangle j \in H$   
**shows**  $\text{exe } (\text{tm-left-until } H \ j) \ (0, tps) = (1, tps)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{exe-tm-left-until-2}$ :  
**assumes**  $j < \text{length } tps$  **and**  $(0, tps) \langle . \rangle j \notin H$   
**shows**  $\text{exe } (\text{tm-left-until } H \ j) \ (0, tps) = (0, tps[j := tps ! j \mid - \ 1])$   
 $\langle \text{proof} \rangle$

We do not show the semantics of  $\text{tm-left-until}$  for the general case, but only for when applied to begin tapes.

**lemma**  $\text{execute-tm-left-until-less}$ :  
**assumes**  $j < \text{length } tps$  **and**  $\text{begin-tape } H \ (tps ! j)$  **and**  $t \leq tps \text{ :\#} j$   
**shows**  $\text{execute } (\text{tm-left-until } H \ j) \ (0, tps) \ t = (0, tps[j := tps ! j \mid - \ t])$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{execute-tm-left-until}$ :  
**assumes**  $j < \text{length } tps$  **and**  $\text{begin-tape } H \ (tps ! j)$   
**shows**  $\text{execute } (\text{tm-left-until } H \ j) \ (0, tps) \ (\text{Suc } (tps \text{ :\#} j)) = (1, tps[j := tps ! j \mid \# = \ 0])$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{transits-tm-left-until}$ :  
**assumes**  $j < \text{length } tps$  **and**  $\text{begin-tape } H \ (tps ! j)$   
**shows**  $\text{transits } (\text{tm-left-until } H \ j) \ (0, tps) \ (\text{Suc } (tps \text{ :\#} j)) \ (1, tps[j := tps ! j \mid \# = \ 0])$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{transforms-tm-left-until}$ :  
**assumes**  $j < \text{length } tps$  **and**  $\text{begin-tape } H \ (tps ! j)$   
**shows**  $\text{transforms } (\text{tm-left-until } H \ j) \ tps \ (\text{Suc } (tps \text{ :\#} j)) \ (tps[j := tps ! j \mid \# = \ 0])$   
 $\langle \text{proof} \rangle$

The most common case is  $H = \{\triangleright\}$ , which means the Turing machine moves the tape head left to the closest start symbol. On clean tapes it moves the tape head to the leftmost cell of the tape.

**definition**  $\text{tm-start} :: \text{tapeidx} \Rightarrow \text{machine}$  **where**  
 $\text{tm-start} \equiv \text{tm-left-until } \{1\}$

**lemma**  $\text{tm-start-tm}$ :  
**assumes**  $k \geq 2$  **and**  $G \geq 4$   
**shows**  $\text{turing-machine } k \ G \ (\text{tm-start } j)$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-start*:  
**assumes**  $j < \text{length } tps$  **and** *clean-tape* ( $tps ! j$ )  
**shows** *transforms* (*tm-start*  $j$ )  $tps$  (*Suc* ( $tps :\# : j$ )) ( $tps[j := tps ! j | \# = | 0]$ )  
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-startI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$  **and** *clean-tape* ( $tps ! j$ )  
**and**  $t = \text{Suc } (tps :\# : j)$   
**and**  $tps' = tps[j := tps ! j | \# = | 0]$   
**shows** *transforms* (*tm-start*  $j$ )  $tps$   $t$   $tps'$   
 $\langle \text{proof} \rangle$

The next Turing machine is the first instance in which we use the ;; operator with concrete Turing machines. It is also the first time we use the proof method *tform* for *transforms*. The TM performs a “carriage return” on a clean tape, that is, it moves to the first non-start symbol.

**definition** *tm-cr* :: *tapeidx*  $\Rightarrow$  *machine* **where**  
 $tm\text{-}cr\ j \equiv tm\text{-}start\ j\ ;;\ tm\text{-}right\ j$

**lemma** *tm-cr-tm*:  $k \geq 2 \implies G \geq 4 \implies \text{turing-machine } k\ G\ (tm\text{-}cr\ j)$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-crI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$   
**and** *clean-tape* ( $tps ! j$ )  
**and**  $t = tps :\# : j + 2$   
**and**  $tps' = tps[j := tps ! j | \# = | 1]$   
**shows** *transforms* (*tm-cr*  $j$ )  $tps$   $t$   $tps'$   
 $\langle \text{proof} \rangle$

## 2.4.7 Erasing a tape

The next Turing machine overwrites all but the start symbol with blanks. It first performs a carriage return and then writes blanks until it reaches a blank. This only works as intended if there are no gaps, that is, blanks between non-blank symbols.

**definition** *tm-erase* :: *tapeidx*  $\Rightarrow$  *machine* **where**  
 $tm\text{-}erase\ j \equiv tm\text{-}cr\ j\ ;;\ tm\text{-}const\text{-}until\ j\ j\ \{\square\}\ \square$

**lemma** *tm-erase-tm*:  $G \geq 4 \implies 0 < j \implies j < k \implies \text{turing-machine } k\ G\ (tm\text{-}erase\ j)$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-eraseI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$   
**and** *proper-symbols*  $zs$   
**and**  $tps\ ::\ j = \lfloor zs \rfloor$   
**and**  $t = tps :\# : j + \text{length } zs + 3$   
**and**  $tps' = tps[j := (\lfloor \square \rfloor, \text{Suc } (\text{length } zs))]$   
**shows** *transforms* (*tm-erase*  $j$ )  $tps$   $t$   $tps'$   
 $\langle \text{proof} \rangle$

The next TM returns to the leftmost blank symbol after erasing the tape.

**definition** *tm-erase-cr* :: *tapeidx*  $\Rightarrow$  *machine* **where**  
 $tm\text{-}erase\text{-}cr\ j \equiv tm\text{-}erase\ j\ ;;\ tm\text{-}cr\ j$

**lemma** *tm-erase-cr-tm*:  
**assumes**  $G \geq 4$  **and**  $0 < j$  **and**  $j < k$   
**shows** *turing-machine*  $k\ G\ (tm\text{-}erase\text{-}cr\ j)$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-erase-crI* [*transforms-intros*]:  
**assumes**  $j < \text{length } tps$   
**and** *proper-symbols*  $zs$

```

and tps ::= j = ⌊zs⌋
and t = tps :#: j + 2 * length zs + 6
and tps' = tps[j := (⌊⌋, 1)]
shows transforms (tm-erase-cr j) tps t tps'
⟨proof⟩

```

## 2.4.8 Writing a symbol sequence

The Turing machine in this section writes a hard-coded symbol sequence to a tape. It is like *tm-write-repeat* except with an arbitrary symbol sequence.

```

fun tm-print :: tapeidx ⇒ symbol list ⇒ machine where
  tm-print j [] = [] |
  tm-print j (z # zs) = tm-char j z ;; tm-print j zs

```

```

lemma tm-print-tm:
assumes 0 < j and j < k and G ≥ 4 and ∀ i < length zs. zs ! i < G
shows turing-machine k G (tm-print j zs)
⟨proof⟩

```

The result of writing the symbols *zs* to a tape *tp*:

```

definition inscribe :: tape ⇒ symbol list ⇒ tape where
  inscribe tp zs ≡
    (λi. if snd tp ≤ i ∧ i < snd tp + length zs then zs ! (i - snd tp) else fst tp i,
     snd tp + length zs)

```

```

lemma inscribe-Nil: inscribe tp [] = tp
⟨proof⟩

```

```

lemma inscribe-Cons: inscribe ((fst tp)(snd tp := z), Suc (snd tp)) zs = inscribe tp (z # zs)
⟨proof⟩

```

```

lemma inscribe-contents: inscribe (⌊ys⌋, Suc (length ys)) zs = (⌊ys @ zs⌋, Suc (length ys + length zs))
  (is ?lhs = ?rhs)
⟨proof⟩

```

```

lemma inscribe-contents-Nil: inscribe (⌊⌋, Suc 0) zs = (⌊zs⌋, Suc (length zs))
⟨proof⟩

```

```

lemma transforms-tm-print:
assumes j < length tps
shows transforms (tm-print j zs) tps (length zs) (tps[j := inscribe (tps ! j) zs])
⟨proof⟩

```

```

lemma transforms-tm-printI [transforms-intros]:
assumes j < length tps and tps' = (tps[j := inscribe (tps ! j) zs])
shows transforms (tm-print j zs) tps (length zs) tps'
⟨proof⟩

```

## 2.4.9 Setting the tape contents to a symbol sequence

The following Turing machine erases the tape, then prints a hard-coded symbol sequence, and then performs a carriage return. It thus sets the tape contents to the symbol sequence.

```

definition tm-set :: tapeidx ⇒ symbol list ⇒ machine where
  tm-set j zs ≡ tm-erase-cr j ;; tm-print j zs ;; tm-cr j

```

```

lemma tm-set-tm:
assumes 0 < j and j < k and G ≥ 4 and ∀ i < length zs. zs ! i < G
shows turing-machine k G (tm-set j zs)
⟨proof⟩

```

```

lemma transforms-tm-setI [transforms-intros]:
assumes j < length tps

```

```

and clean-tape (tps ! j)
and proper-symbols ys
and proper-symbols zs
and tps :: j = |ys|
and t = 8 + tps :# : j + 2 * length ys + Suc (2 * length zs)
and tps' = tps[j := (|zs|, 1)]
shows transforms (tm-set j zs) tps t tps'
⟨proof⟩

```

## 2.4.10 Comparing two tapes

The next Turing machine compares the contents of two tapes  $j_1$  and  $j_2$  and writes to tape  $j_3$  either a  $\mathbf{1}$  or a  $\square$  depending on whether the tapes are equal or not. The next command does all the work. It scans both tapes left to right and halts if it encounters a blank on both tapes, which means the tapes are equal, or two different symbols, which means the tapes are unequal. It only works for contents without blanks.

**definition** *cmd-cmp* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *command* **where**

```

cmd-cmp j1 j2 j3 rs  $\equiv$ 
  if rs ! j1  $\neq$  rs ! j2
  then (1, map ( $\lambda i.$  (if i = j3 then  $\square$  else rs ! i, Stay)) [0..length rs])
  else if rs ! j1 =  $\square$   $\vee$  rs ! j2 =  $\square$ 
  then (1, map ( $\lambda i.$  (if i = j3 then  $\mathbf{1}$  else rs ! i, Stay)) [0..length rs])
  else (0, map ( $\lambda i.$  (rs ! i, if i = j1  $\vee$  i = j2 then Right else Stay)) [0..length rs])

```

**lemma** *sem-cmd-cmp1*:

```

assumes length tps = k
and j1 < k and j2 < k and j3 < k
and tps :: j1  $\neq$  tps :: j2
shows sem (cmd-cmp j1 j2 j3) (0, tps) = (1, tps[j3 := tps ! j3 |:=|  $\square$ ])
⟨proof⟩

```

**lemma** *sem-cmd-cmp2*:

```

assumes length tps = k
and j1 < k and j2 < k and j3 < k
and tps :: j1 = tps :: j2 and tps :: j1 =  $\square$   $\vee$  tps :: j2 =  $\square$ 
shows sem (cmd-cmp j1 j2 j3) (0, tps) = (1, tps[j3 := tps ! j3 |:=|  $\mathbf{1}$ ])
⟨proof⟩

```

**lemma** *sem-cmd-cmp3*:

```

assumes length tps = k
and j2  $\neq$  j3 and j1  $\neq$  j3 and j1 < k and j2 < k and j3 < k
and tps :: j1 = tps :: j2 and tps :: j1  $\neq$   $\square$   $\wedge$  tps :: j2  $\neq$   $\square$ 
shows sem (cmd-cmp j1 j2 j3) (0, tps) = (0, tps[j1 := tps ! j1 |+| 1, j2 := tps ! j2 |+| 1])
⟨proof⟩

```

**definition** *tm-cmp* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-cmp j1 j2 j3  $\equiv$  [cmd-cmp j1 j2 j3]

```

**lemma** *tm-cmp-tm*:

```

assumes k  $\geq$  2 and j3 > 0 and G  $\geq$  4
shows turing-machine k G (tm-cmp j1 j2 j3)
⟨proof⟩

```

**lemma** *exe-cmd-cmp1*:

```

assumes length tps = k
and j1 < k and j2 < k and j3 < k
and tps :: j1  $\neq$  tps :: j2
shows exe (tm-cmp j1 j2 j3) (0, tps) = (1, tps[j3 := tps ! j3 |:=|  $\square$ ])
⟨proof⟩

```

**lemma** *exe-cmd-cmp2*:

```

assumes length tps = k
and j1 < k and j2 < k and j3 < k
and tps :: j1 = tps :: j2 and tps :: j1 =  $\square$   $\vee$  tps :: j2 =  $\square$ 

```

**shows**  $exe (tm\text{-}cmp\ j1\ j2\ j3) (0, tps) = (1, tps[j3 := tps ! j3 \mid := \mathbf{1}])$   
 ⟨proof⟩

**lemma** *exe-cmd-cmp3*:

**assumes**  $length\ tps = k$   
**and**  $j2 \neq j3$  **and**  $j1 \neq j3$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $j3 < k$   
**and**  $tps \ ::\ j1 = tps \ ::\ j2$  **and**  $tps \ ::\ j1 \neq \square \wedge tps \ ::\ j2 \neq \square$   
**shows**  $exe (tm\text{-}cmp\ j1\ j2\ j3) (0, tps) = (0, tps[j1 := tps ! j1 \mid +\ 1, j2 := tps ! j2 \mid +\ 1])$   
 ⟨proof⟩

**lemma** *execute-tm-cmp-eq*:

**fixes**  $tps \ ::\ tape\ list$   
**assumes**  $length\ tps = k$   
**and**  $j2 \neq j3$  **and**  $j1 \neq j3$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $j3 < k$   
**and** *proper-symbols*  $xs$   
**and**  $tps ! j1 = (\lfloor xs \rfloor, 1)$   
**and**  $tps ! j2 = (\lfloor xs \rfloor, 1)$   
**shows**  $execute (tm\text{-}cmp\ j1\ j2\ j3) (0, tps) (Suc (length\ xs)) =$   
 $(1, tps[j1 := tps ! j1 \mid +\ length\ xs, j2 := tps ! j2 \mid +\ length\ xs, j3 := tps ! j3 \mid := \mathbf{1}])$   
 ⟨proof⟩

**lemma** *ex-contents-neg*:

**assumes** *proper-symbols*  $xs$  **and** *proper-symbols*  $ys$  **and**  $xs \neq ys$   
**shows**  $\exists m. m \leq Suc (\min (length\ xs) (length\ ys)) \wedge \lfloor xs \rfloor\ m \neq \lfloor ys \rfloor\ m$   
 ⟨proof⟩

**lemma** *execute-tm-cmp-neg*:

**fixes**  $tps \ ::\ tape\ list$   
**assumes**  $length\ tps = k$   
**and**  $j1 \neq j2$  **and**  $j2 \neq j3$  **and**  $j1 \neq j3$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $j3 < k$   
**and** *proper-symbols*  $xs$   
**and** *proper-symbols*  $ys$   
**and**  $xs \neq ys$   
**and**  $tps ! j1 = (\lfloor xs \rfloor, 1)$   
**and**  $tps ! j2 = (\lfloor ys \rfloor, 1)$   
**and**  $m = (LEAST\ m. m \leq Suc (\min (length\ xs) (length\ ys)) \wedge \lfloor xs \rfloor\ m \neq \lfloor ys \rfloor\ m)$   
**shows**  $execute (tm\text{-}cmp\ j1\ j2\ j3) (0, tps) m =$   
 $(1, tps[j1 := tps ! j1 \mid +\ (m - 1), j2 := tps ! j2 \mid +\ (m - 1), j3 := tps ! j3 \mid := \square])$   
 ⟨proof⟩

**lemma** *transforms-tm-cmpI* [*transforms-intros*]:

**fixes**  $tps \ ::\ tape\ list$   
**assumes**  $length\ tps = k$   
**and**  $j1 \neq j2$  **and**  $j2 \neq j3$  **and**  $j1 \neq j3$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $j3 < k$   
**and** *proper-symbols*  $xs$   
**and** *proper-symbols*  $ys$   
**and**  $tps ! j1 = (\lfloor xs \rfloor, 1)$   
**and**  $tps ! j2 = (\lfloor ys \rfloor, 1)$   
**and**  $t = Suc (\min (length\ xs) (length\ ys))$   
**and**  $b = (if\ xs = ys\ then\ \mathbf{1}\ else\ \square)$   
**and**  $m =$   
 $(if\ xs = ys$   
 $\ then\ Suc (length\ xs)$   
 $\ else\ (LEAST\ m. m \leq Suc (\min (length\ xs) (length\ ys)) \wedge \lfloor xs \rfloor\ m \neq \lfloor ys \rfloor\ m))$   
**and**  $tps' = tps[j1 := (\lfloor xs \rfloor, m), j2 := (\lfloor ys \rfloor, m), j3 := tps ! j3 \mid := b]$   
**shows**  $transforms (tm\text{-}cmp\ j1\ j2\ j3) tps\ t\ tps'$   
 ⟨proof⟩

The next Turing machine extends *tm-cmp* by a carriage return on tapes  $j_1$  and  $j_2$ , ensuring that the next command finds the tape heads in a well-specified position. This makes the TM easier to reuse.

**definition** *tm-equals*  $\ ::\ tapeidx \Rightarrow tapeidx \Rightarrow tapeidx \Rightarrow machine$  **where**

$tm\text{-}equals\ j1\ j2\ j3 \equiv tm\text{-}cmp\ j1\ j2\ j3 \ ;\ ;\ tm\text{-}cr\ j1 \ ;\ ;\ tm\text{-}cr\ j2$

**lemma** *tm-equals-tm*:  
**assumes**  $k \geq 2$  **and**  $j_3 > 0$  **and**  $G \geq 4$   
**shows** *turing-machine*  $k$   $G$  (*tm-equals*  $j_1$   $j_2$   $j_3$ )  
 $\langle$ *proof* $\rangle$

We analyze the behavior of *tm-equals* inside a locale. This is how we will typically proceed for Turing machines that are composed of more than two TMs. The locale is parameterized by the TM's parameters, which in the present case means the three tape indices  $j_1$ ,  $j_2$ , and  $j_3$ . Inside the locale the TM is decomposed such that proofs of *transforms* only involve two TMs combined by one of the three control structures (sequence, branch, loop). In the current example we have three TMs named *tm1*, *tm2*, *tm3*, where *tm3* is just *tm-equals*. Furthermore there will be lemmas *tm1*, *tm2*, *tm3* describing, in terms of *transforms*, the behavior of the respective TMs. For this we define three tape lists *tps1*, *tps2*, *tps3*.

This naming scheme creates many name clashes for things that only have a single use. That is the reason for the encapsulation in a locale.

Afterwards this locale is interpreted, just once in lemma *transforms-tm-equalsI*, to prove the semantics and running time of *tm-equals*.

**locale** *turing-machine-equals* =  
**fixes**  $j_1$   $j_2$   $j_3$  :: *tapeidx*  
**begin**

**definition**  $tm1 \equiv tm\text{-}cmp\ j_1\ j_2\ j_3$   
**definition**  $tm2 \equiv tm1\ ;\ ;\ tm\text{-}cr\ j_1$   
**definition**  $tm3 \equiv tm2\ ;\ ;\ tm\text{-}cr\ j_2$

**lemma** *tm3-eq-tm-equals*:  $tm3 = tm\text{-}equals\ j_1\ j_2\ j_3$   
 $\langle$ *proof* $\rangle$

**context**

**fixes**  $tps0$  :: *tape list* **and**  $k\ t\ b$  :: *nat* **and**  $xs\ ys$  :: *symbol list*  
**assumes**  $jk$  [*simp*]:  $length\ tps0 = k$   $j_1 \neq j_2$   $j_2 \neq j_3$   $j_1 \neq j_3$   $j_1 < k$   $j_2 < k$   $j_3 < k$   
**and** *proper*: *proper-symbols*  $xs$  *proper-symbols*  $ys$   
**and**  $t$ :  $t = Suc\ (min\ (length\ xs)\ (length\ ys))$   
**and**  $b$ :  $b = (if\ xs = ys\ then\ 3\ else\ 0)$

**assumes**  $tps0$ :  
 $tps0\ !\ j_1 = (\lfloor xs \rfloor, 1)$   
 $tps0\ !\ j_2 = (\lfloor ys \rfloor, 1)$

**begin**

**definition**  $m \equiv$   
 $(if\ xs = ys$   
 $then\ Suc\ (length\ xs)$   
 $else\ (LEAST\ m.\ m \leq Suc\ (min\ (length\ xs)\ (length\ ys)) \wedge \lfloor xs \rfloor\ m \neq \lfloor ys \rfloor\ m))$

**lemma** *m-gr-0*:  $m > 0$   
 $\langle$ *proof* $\rangle$

**lemma** *m-le-t*:  $m \leq t$   
 $\langle$ *proof* $\rangle$

**definition**  $tps1 \equiv tps0[j_1 := (\lfloor xs \rfloor, m), j_2 := (\lfloor ys \rfloor, m), j_3 := tps0\ !\ j_3\ |:=| b]$

**lemma** *tm1 [transforms-intros]*: *transforms*  $tm1$   $tps0$   $t$   $tps1$   
 $\langle$ *proof* $\rangle$

**definition**  $tps2 \equiv tps0[j_1 := (\lfloor xs \rfloor, 1), j_2 := (\lfloor ys \rfloor, m), j_3 := tps0\ !\ j_3\ |:=| b]$

**lemma** *tm2*:  
**assumes**  $ttt = t + m + 2$   
**shows** *transforms*  $tm2$   $tps0$   $ttt$   $tps2$   
 $\langle$ *proof* $\rangle$

**lemma** *tm2'* [transforms-intros]: transforms *tm2 tps0 (2 \* t + 2) tps2*  
 ⟨proof⟩

**definition** *tps3*  $\equiv$  *tps0*[*j1* := ( $\lfloor xs \rfloor$ , 1), *j2* := ( $\lfloor ys \rfloor$ , 1), *j3* := *tps0* ! *j3* |:=| *b*]

**lemma** *tm3*:  
 assumes *tmt* =  $2 * t + m + 4$   
 shows transforms *tm3 tps0 tmt tps3*  
 ⟨proof⟩

**definition** *tps3'*  $\equiv$  *tps0*[*j3* := *tps0* ! *j3* |:=| *b*]

**lemma** *tm3'*: transforms *tm3 tps0 (3 \* min (length xs) (length ys) + 7) tps3'*  
 ⟨proof⟩

**end**

**end**

**lemma** *transforms-tm-equalsI* [transforms-intros]:  
 fixes *j1 j2 j3* :: *tapeidx*  
 fixes *tps tps'* :: *tape list* **and** *k* :: *nat* **and** *xs ys* :: *symbol list* **and** *b* :: *symbol*  
 assumes *length tps* = *k* *j1*  $\neq$  *j2* *j2*  $\neq$  *j3* *j1*  $\neq$  *j3* *j1* < *k* *j2* < *k* *j3* < *k*  
**and** *proper-symbols xs proper-symbols ys*  
**and** *b* = (if *xs* = *ys* then 1 else  $\square$ )  
 assumes  
   *tps* ! *j1* = ( $\lfloor xs \rfloor$ , 1)  
   *tps* ! *j2* = ( $\lfloor ys \rfloor$ , 1)  
 assumes *tmt* =  $3 * \min (\text{length } xs) (\text{length } ys) + 7$   
 assumes *tps'* = *tps*  
   [*j3* := *tps* ! *j3* |:=| *b*]  
 shows transforms (*tm-equals j1 j2 j3*) *tps tmt tps'*  
 ⟨proof⟩

## 2.4.11 Computing the identity function

In order to compute the identity function, a Turing machine can just copy the input tape to the output tape:

**definition** *tm-id* :: *machine* **where**  
*tm-id*  $\equiv$  *tm-cp-until* 0 1 { $\square$ }

**lemma** *tm-id-tm*:  
 assumes  $1 < k$  **and**  $G \geq 4$   
 shows *turing-machine k G tm-id*  
 ⟨proof⟩

**lemma** *transforms-tm-idI*:  
 fixes *zs* :: *symbol list* **and** *k* :: *nat* **and** *tps* :: *tape list*  
 assumes  $1 < k$   
**and** *proper-symbols zs*  
**and** *tps* = *snd (start-config k zs)*  
**and** *tps'* = *tps*[0 := ( $\lfloor zs \rfloor$ , (*Suc* (length *zs*))), 1 := ( $\lfloor zs \rfloor$ , (*Suc* (length *zs*)))]  
 shows transforms *tm-id tps (Suc (Suc (length zs))) tps'*  
 ⟨proof⟩

The identity function is computable with a time bound of  $n + 2$ .

**lemma** *computes-id*: *computes-in-time 2 tm-id id* ( $\lambda n. \text{Suc } (\text{Suc } n)$ )  
 ⟨proof⟩

**end**

## 2.5 Memorizing in states

```

theory Memorizing
  imports Elementary
begin

```

Some Turing machines are best described by allowing them to memorize values in their states. For example, a TM that adds two binary numbers could memorize the carry bit in states. In the textbook definition of TMs, with arbitrary state space, this can be represented by a state space of the form  $Q \times \{0, 1\}$ , where 0 and 1 represent the memorized values. In our simplified definition of TMs, where the state space is an interval of natural numbers, this does not work. However, there is a workaround. Since we can have arbitrarily many tapes, we can make the TM store this value on an additional tape. Such a memorization tape could be used to write/read a symbol representing the memorized value. The tape head would never move on such a tape. The behavior of the TM can then depend on the memorized value.

By adding several such tapes we can even have more than one value stored simultaneously as well. However, this method increases the number of tapes, and one part of the proof of the Cook-Levin theorem is showing that every TM can be simulated on a two-tape TM (see Chapter 5.3). How to remove such memorization tapes again without changing the behavior of the TM is the subject of this section.

The straightforward idea is to multiply the states by the number of possible values. So if the original TM has  $Q$  non-halting states and memorizes  $G$  different values, the new TM has  $Q \cdot G$  non-halting states. It would be natural to map a pair  $(q, g)$  of state and memorized value to  $q \cdot G + g$  or to  $g \cdot Q + q$ . However, there is a small technical problem.

The memorization tape is initialized, like all tapes in a start configuration, with the head on the leftmost cell, which contains the start symbol. Thus the initially memorized value is the number 1 representing  $\triangleright$ . The new TM must start in the start state, which we have fixed at 0. Thus the state-value pair  $(0, 1)$  must be mapped to 0, which neither of the two natural mappings does. Our workaround is to use the mapping  $(q, g) \mapsto ((g - 1) \bmod G) \cdot Q + q$ .

The following function maps a Turing machine  $M$  that memorizes one value from  $\{0, \dots, G - 1\}$  on its last tape to a TM that has one tape less, has  $G$  times the number of non-halting states, and behaves just like  $M$ . The name “cartesian” for this function is just a funny term I made up.

**definition** *cartesian* :: *machine*  $\Rightarrow$  *nat*  $\Rightarrow$  *machine* **where**

```

cartesian M G  $\equiv$ 
  concat
  (map ( $\lambda h$ . map ( $\lambda q$  rs.
    let ( $q'$ , as) = (M ! q) (rs @ [(h + 1) mod G])
    in (if  $q' = \text{length } M$  then  $G * \text{length } M$  else (fst (last as) + G - 1) mod G * length M +  $q'$ ,
      butlast as))
    [0..length M])
  [0.. $G$ ])

```

**lemma** *length-concat-const*:

```

assumes  $\bigwedge h$ . length (f h) = c
shows length (concat (map f [0.. $G$ ])) =  $G * c$ 
<proof>

```

**lemma** *length-cartesian*: *length* (*cartesian* M G) =  $G * \text{length } M$

<*proof*>

**lemma** *concat-nth*:

```

assumes  $\bigwedge h$ . length (f h) = c
and xs = concat (map f [0.. $G$ ])
and  $h < G$ 
and  $q < c$ 
shows xs ! ( $h * c + q$ ) = f h ! q
<proof>

```

**lemma** *cartesian-at*:

```

assumes  $M' = \text{cartesian } M \ b$  and  $h < b$  and  $q < \text{length } M$ 
shows ( $M' ! (h * \text{length } M + q)$ ) rs =
  (let ( $q'$ , as) = (M ! q) (rs @ [(h + 1) mod b])

```

in (if  $q' = \text{length } M$  then  $b * \text{length } M$  else  $(\text{fst } (\text{last } as) + b - 1) \bmod b * \text{length } M + q'$ ,  
butlast as)  
(proof)

**lemma** *concat-nth-ex*:

**assumes**  $\bigwedge h. \text{length } (f h) = c$   
**and**  $xs = \text{concat } (\text{map } f [0..<G])$   
**and**  $j < G * c$   
**shows**  $\exists i h. i < c \wedge h < G \wedge xs ! j = f h ! i$   
 (proof)

The cartesian TM has one tape less than the original TM.

**lemma** *cartesian-num-tapes*:

**assumes** *turing-machine* (Suc k) G M  
**and**  $M' = \text{cartesian } M b$   
**and**  $\text{length } rs = k$   
**and**  $q' < \text{length } M'$   
**shows**  $\text{length } (\text{snd } ((M' ! q') rs)) = k$   
 (proof)

The cartesian TM of a TM with alphabet  $G$  also has the alphabet  $G$  provided it memorizes at most  $G$  values.

**lemma** *cartesian-tm*:

**assumes** *turing-machine* (Suc k) G M  
**and**  $M' = \text{cartesian } M b$   
**and**  $k \geq 2$   
**and**  $b \leq G$   
**and**  $b > 0$   
**shows** *turing-machine* k G M'  
 (proof)

A special case of the previous lemma is  $b = G$ :

**corollary** *cartesian-tm'*:

**assumes** *turing-machine* (Suc k) G M  
**and**  $M' = \text{cartesian } M G$   
**and**  $k \geq 2$   
**shows** *turing-machine* k G M'  
 (proof)

A cartesian TM assumes essentially the same configurations the original machine does, except that it has one tape less and the states have a greater number. We call these configurations “squished”, another fancy made-up term alluding to the removal of one tape.

**definition** *squish* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{config} \Rightarrow \text{config}$  **where**

*squish* G Q cfg  $\equiv$   
 let  $(q, tps) = \text{cfg}$   
 in (if  $q \geq Q$  then  $G * Q$  else  $(| \cdot | (\text{last } tps) + G - 1) \bmod G * Q + q$ , butlast tps)

**lemma** *squish*:

*squish* G Q cfg =  
 (if  $\text{fst } \text{cfg} \geq Q$  then  $G * Q$  else  $(| \cdot | (\text{last } (\text{snd } \text{cfg})) + G - 1) \bmod G * Q + \text{fst } \text{cfg}$ , butlast (snd cfg))  
 (proof)

**lemma** *squish-head-pos*:

**assumes**  $\|\text{cfg}\| > 2$   
**shows** *squish* G Q cfg  $\langle \# \rangle 0 = \text{cfg} \langle \# \rangle 0$   
**and** *squish* G Q cfg  $\langle \# \rangle 1 = \text{cfg} \langle \# \rangle 1$   
 (proof)

**lemma** *mod-less*:

**fixes** q Q h G :: nat  
**assumes**  $q < Q$  **and**  $0 < G$

**shows**  $h \text{ mod } G * Q + q < G * Q$   
 ⟨proof⟩

**lemma** *squish-halt-state*:

**assumes**  $G > 0$  **and**  $\text{fst } \text{cfg} \leq Q$   
**shows**  $\text{fst } (\text{squish } G \ Q \ \text{cfg}) = G * Q \iff \text{fst } \text{cfg} = Q$   
 ⟨proof⟩

**lemma** *butlast-replicate*:  $\text{butlast } (\text{replicate } k \ x) = \text{replicate } (k - \text{Suc } 0) \ x$   
 ⟨proof⟩

**lemma** *squish-start-config*:  $G \geq 4 \implies k \geq 2 \implies \text{squish } G \ Q \ (\text{start-config } (\text{Suc } k) \ zs) = \text{start-config } k \ zs$   
 ⟨proof⟩

The cartesian Turing machine only works properly if the original TM never moves its head on the last tape. We call a tape of a TM  $M$  *immobile* if  $M$  never moves the head on the tape.

**definition** *immobile* ::  $\text{machine} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**

$\text{immobile } M \ j \ k \equiv \forall q \text{ rs. } q < \text{length } M \longrightarrow \text{length } \text{rs} = k \longrightarrow (M \ ! \ q) \ \text{rs} \ [\sim] \ j = \text{Stay}$

**lemma** *immobileI* [*intro*]:

**assumes**  $\bigwedge q \text{ rs. } q < \text{length } M \implies \text{length } \text{rs} = k \implies (M \ ! \ q) \ \text{rs} \ [\sim] \ j = \text{Stay}$   
**shows**  $\text{immobile } M \ j \ k$   
 ⟨proof⟩

If the head never moves on tape  $k$ , the head will stay in position 0.

**lemma** *immobile-head-pos-proper*:

**assumes**  $\text{proper-machine } (\text{Suc } k) \ M$   
**and**  $\text{immobile } M \ k \ (\text{Suc } k)$   
**and**  $\|\text{cfg}\| = \text{Suc } k$   
**shows**  $\text{execute } M \ \text{cfg } t \ <\#\> \ k = \text{cfg } <\#\> \ k$   
 ⟨proof⟩

**lemma** *immobile-head-pos*:

**assumes**  $\text{turing-machine } (\text{Suc } k) \ G \ M$   
**and**  $\text{immobile } M \ k \ (\text{Suc } k)$   
**and**  $\|\text{cfg}\| = \text{Suc } k$   
**shows**  $\text{execute } M \ \text{cfg } t \ <\#\> \ k = \text{cfg } <\#\> \ k$   
 ⟨proof⟩

Sequentially combining two Turing machines with an immobile tape yields a Turing machine with the same immobile tape.

**lemma** *immobile-sequential*:

**assumes**  $\text{turing-machine } k \ G \ M1$   
**and**  $\text{turing-machine } k \ G \ M2$   
**and**  $\text{immobile } M1 \ j \ k$   
**and**  $\text{immobile } M2 \ j \ k$   
**shows**  $\text{immobile } (M1 \ ;; \ M2) \ j \ k$   
 ⟨proof⟩

A loop also keeps a tape immobile.

**lemma** *immobile-loop*:

**assumes**  $\text{turing-machine } k \ G \ M1$   
**and**  $\text{turing-machine } k \ G \ M2$   
**and**  $\text{immobile } M1 \ j \ k$   
**and**  $\text{immobile } M2 \ j \ k$   
**and**  $j < k$   
**shows**  $\text{immobile } (\text{WHILE } M1 \ ; \ \text{cond } \text{DO } M2 \ \text{DONE}) \ j \ k$   
 ⟨proof⟩

An immobile tape stays immobile when further tapes are appended. We only need this for the special case of two-tape Turing machines.

**lemma** *immobile-append-tapes*:

**assumes**  $j < k$  **and**  $j > 1$  **and**  $k \geq 2$  **and** *turing-machine 2 G M*  
**shows** *immobile (append-tapes 2 k M) j k*  
*<proof>*

For the elementary Turing machines we introduced in Section 2.4 all tapes are immobile but the ones given as parameters.

**lemma** *immobile-tm-trans-until*:  
**assumes**  $j \neq j1$  **and**  $j \neq j2$  **and**  $j < k$   
**shows** *immobile (tm-trans-until j1 j2 H f) j k*  
*<proof>*

**lemma** *immobile-tm-ltrans-until*:  
**assumes**  $j \neq j1$  **and**  $j \neq j2$  **and**  $j < k$   
**shows** *immobile (tm-ltrans-until j1 j2 H f) j k*  
*<proof>*

**lemma** *immobile-tm-left-until*:  
**assumes**  $j \neq j'$  **and**  $j < k$   
**shows** *immobile (tm-left-until H j') j k*  
*<proof>*

**lemma** *immobile-tm-start*:  
**assumes**  $j \neq j'$  **and**  $j < k$   
**shows** *immobile (tm-start j') j k*  
*<proof>*

**lemma** *immobile-tm-write*:  
**assumes**  $j < k$   
**shows** *immobile (tm-write j' h) j k*  
*<proof>*

**lemma** *immobile-tm-write-many*:  
**assumes**  $j < k$   
**shows** *immobile (tm-write-many J h) j k*  
*<proof>*

**lemma** *immobile-tm-right*:  
**assumes**  $j \neq j'$  **and**  $j < k$   
**shows** *immobile (tm-right j') j k*  
*<proof>*

**lemma** *immobile-tm-rtrans*:  
**assumes**  $j \neq j'$  **and**  $j < k$   
**shows** *immobile (tm-rtrans j' f) j k*  
*<proof>*

**lemma** *immobile-tm-left*:  
**assumes**  $j \neq j'$  **and**  $j < k$   
**shows** *immobile (tm-left j') j k*  
*<proof>*

**lemma** *mod-inc-dec*:  $(h::nat) < G \implies ((h + G - 1) \bmod G + 1) \bmod G = h$   
*<proof>*

**lemma** *last-length*:  $\text{length } xs = \text{Suc } k \implies \text{last } xs = xs ! k$   
*<proof>*

The tapes used for memorizing the values have blank symbols in every cell but possibly for the leftmost cell. In keeping with funny names, we call such tapes *onesie* tapes.

**definition** *onesie* :: *symbol*  $\Rightarrow$  *tape* ( $\langle [-] \rangle$ ) **where**  
 $[h] \equiv (\lambda x. \text{if } x = 0 \text{ then } h \text{ else } \square, 0)$

**lemma** *onesie-1*:  $[>] = (\llbracket \rrbracket, 0)$

*<proof>*

**lemma onesie-read** [*simp*]:  $|\cdot| \lceil h \rceil = h$   
*<proof>*

**lemma onesie-write**:  $\lceil x \rceil | := | y = \lceil y \rceil$   
*<proof>*

**lemma act-onesie**: *act* (*h*, *Stay*)  $\lceil x \rceil = \lceil h \rceil$   
*<proof>*

We now consider the semantics of cartesian Turing machines. Roughly speaking, a cartesian TM assumes the squished configurations of the original TM. A crucial assumption here is that the original TM only ever memorizes a symbol from a certain range of symbols, with one relaxation: when switching to the halting state, any symbol may be written to the memorization tape. The reason is that there is only one halting state even for the cartesian TM, and thus the halting state is not subject to the mapping of states implemented by the *cartesian* operation.

In the following lemma,  $\lceil \triangleright \rceil$  is the memorization tape. It has the start symbol because in the start configuration all tapes have the start symbol in the leftmost cell.

**lemma cartesian-execute**:

**assumes** *turing-machine* (*Suc k*) *G M*  
**and** *immobile* *M k* (*Suc k*)  
**and**  $k \geq 2$   
**and**  $b > 0$   
**and**  $\text{length } tps = k$   
**and**  $\bigwedge t. \text{execute } M (0, tps @ \lceil \triangleright \rceil) t <.> k < b \vee \text{fst } (\text{execute } M (0, tps @ \lceil \triangleright \rceil) t) = \text{length } M$   
**shows**  $\text{execute } (\text{cartesian } M b) (0, tps) t =$   
 $\text{squish } b (\text{length } M) (\text{execute } M (0, tps @ \lceil \triangleright \rceil) t)$   
*<proof>*

One assumption of the previous lemma is that the memorization tape can only contain a symbol from a certain range (except in the halting state). One way to achieve this is for the Turing machine to only ever write a symbol from that range to the memorization tape (or switch to the halting state). Formally:

**definition bounded-write** :: *machine*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *bool* **where**

*bounded-write* *M k b*  $\equiv$   
 $\forall q rs. q < \text{length } M \longrightarrow \text{length } rs = \text{Suc } k \longrightarrow (M ! q) rs \cdot k < b \vee \text{fst } ((M ! q) rs) = \text{length } M$

The advantage of *bounded-write* is that it is a relatively easy to prove property of a Turing machine. With *bounded-write* the previous lemma, *cartesian-execute*, turns into the following one, where the assumption  $b > 0$  becomes  $b > 1$  because initially the memorization tape has the start symbol, represented by the number 1.

**lemma cartesian-execute-onesie**:

**assumes** *turing-machine* (*Suc k*) *G M*  
**and** *immobile* *M k* (*Suc k*)  
**and**  $k \geq 2$   
**and**  $b > 1$   
**and**  $\text{length } tps = k$   
**and** *bounded-write* *M k b*  
**shows**  $\text{execute } (\text{cartesian } M b) (0, tps) t = \text{squish } b (\text{length } M) (\text{execute } M (0, tps @ \lceil \triangleright \rceil) t)$   
*<proof>*

In the following lemma, the term  $\lceil c \rceil$  reflects the fact that in the halting state the memorized symbol can be anything.

**lemma cartesian-transforms-onesie**:

**assumes** *turing-machine* (*Suc k*) *G M*  
**and** *immobile* *M k* (*Suc k*)  
**and**  $k \geq 2$   
**and**  $b > 1$   
**and** *bounded-write* *M k b*  
**and**  $\text{length } tps = k$

**and** transforms  $M$  ( $tps$  @  $[[\triangleright]]$ )  $t$  ( $tps'$  @  $[[c]]$ )  
**shows** transforms (*cartesian*  $M$   $b$ )  $tps$   $t$   $tps'$   
 <proof>

A Turing machine with alphabet  $G$ , when started on a symbol sequence over  $G$ , is guaranteed to only write symbols from  $G$  to any of its tapes, including any memorization tapes. Therefore the last assumption of lemma *cartesian-execute* is satisfied. So in the case of the start configuration we do not need any extra assumptions such as *bounded-write*. This is formalized in the next lemma. The downside is that it can only be applied to “finished” TMs but not to reusable TMs, because these do not usually start in the start state.

**lemma** *cartesian-execute-start-config*:  
**assumes** *turing-machine* ( $Suc$   $k$ )  $G$   $M$   
**and** *immobile*  $M$   $k$  ( $Suc$   $k$ )  
**and**  $k \geq 2$   
**and**  $\forall i < length\ zs.\ zs\ !\ i < G$   
**shows** *execute* (*cartesian*  $M$   $G$ ) (*start-config*  $k$   $zs$ )  $t =$   
*squish*  $G$  ( $length$   $M$ ) (*execute*  $M$  (*start-config* ( $Suc$   $k$ )  $zs$ )  $t$ )  
 <proof>

So far we have only considered single memorization tapes. But of course we can have more than one by iterating the *cartesian* function. Applying this functions once removes the final memorization tape, but leaves others intact, that is, it maintains immobile tapes:

**lemma** *cartesian-immobile*:  
**assumes** *turing-machine* ( $Suc$   $k$ )  $G$   $M$   
**and**  $j < k$   
**and** *immobile*  $M$   $j$  ( $Suc$   $k$ )  
**and**  $M' = cartesian\ M\ G$   
**shows** *immobile*  $M'$   $j$   $k$   
 <proof>

With the next function, *icartesian*, we can strip several memorization tapes off.

**fun** *icartesian* ::  $nat \Rightarrow machine \Rightarrow nat \Rightarrow machine$  **where**  
*icartesian* 0  $M$   $G = M$  |  
*icartesian* ( $Suc$   $k$ )  $M$   $G = icartesian\ k$  (*cartesian*  $M$   $G$ )  $G$

Applying *icartesian* maintains the property of being a Turing machine. We show that only for the special case that all tapes but the input and output tapes are memorization tapes. In this case, we end up with a two-tape machine.

**lemma** *icartesian-tm*:  
**assumes** *turing-machine* ( $k + 2$ )  $G$   $M$   
**and**  $\bigwedge j.\ j < k \implies immobile\ M\ j\ (k + 2)$   
**shows** *turing-machine* 2  $G$  (*icartesian*  $k$   $M$   $G$ )  
 <proof>

At this point we ought to prove something about the semantics of *icartesian*. However, we will only need one specific result, which we can only express at the end of Section 5.1 after we have introduced oblivious Turing machines.

**end**

## 2.6 Composing functions

**theory** *Composing*  
**imports** *Elementary*  
**begin**

For a Turing machine  $M_1$  computing  $f_1$  in time  $T_1$  and a TM  $M_2$  computing  $f_2$  in time  $T_2$  there is a TM  $M$  computing  $f_2 \circ f_1$  in time  $O(T_1(n) + \max_{m \leq T_1(n)} T_2(m))$ . If  $T_1$  is monotone the time bound is  $O(T_1 + T_2 \circ T_1)$ ; if  $T_1$  and  $T_2$  are polynomially bounded the running-time of  $M$  is polynomially bounded, too.

The Turing machines  $M_1$  and  $M_2$  can have both a different alphabet and number of tapes, so generally they cannot be composed by the ; ; operator. To get around this we enlarge the alphabet and prepend

and append tapes, so  $M$  has as many tapes as  $M_1$  and  $M_2$  combined. The following function returns the combined Turing machine  $M$ .

```
definition compose-machines  $k1\ G1\ M1\ k2\ G2\ M2 \equiv$ 
  enlarged  $G1\ (\text{append-tapes } k1\ (k1 + k2)\ M1) \;;$ 
  tm-start  $1 \;;$ 
  tm-cp-until  $1\ k1\ \{\square\} \;;$ 
  tm-erase-cr  $1 \;;$ 
  tm-start  $k1 \;;$ 
  prepend-tapes  $k1\ (\text{enlarged } G2\ M2) \;;$ 
  tm-cr  $(k1 + 1) \;;$ 
  tm-cp-until  $(k1 + 1)\ 1\ \{\square\}$ 
```

```
locale compose =
  fixes  $k1\ k2\ G1\ G2 \::\ \text{nat}$ 
  and  $M1\ M2 \::\ \text{machine}$ 
  and  $T1\ T2 \::\ \text{nat} \Rightarrow \text{nat}$ 
  and  $f1\ f2 \::\ \text{string} \Rightarrow \text{string}$ 
  assumes tm-M1: turing-machine  $k1\ G1\ M1$ 
  and tm-M2: turing-machine  $k2\ G2\ M2$ 
  and computes1: computes-in-time  $k1\ M1\ f1\ T1$ 
  and computes2: computes-in-time  $k2\ M2\ f2\ T2$ 
begin
```

```
definition tm1  $\equiv$  enlarged  $G1\ (\text{append-tapes } k1\ (k1 + k2)\ M1)$ 
definition tm2  $\equiv$  tm1  $\;;$  tm-start  $1$ 
definition tm3  $\equiv$  tm2  $\;;$  tm-cp-until  $1\ k1\ \{\square\}$ 
definition tm4  $\equiv$  tm3  $\;;$  tm-erase-cr  $1$ 
definition tm5  $\equiv$  tm4  $\;;$  tm-start  $k1$ 
definition tm56  $\equiv$  prepend-tapes  $k1\ (\text{enlarged } G2\ M2)$ 
definition tm6  $\equiv$  tm5  $\;;$  tm56
definition tm7  $\equiv$  tm6  $\;;$  tm-cr  $(k1 + 1)$ 
definition tm8  $\equiv$  tm7  $\;;$  tm-cp-until  $(k1 + 1)\ 1\ \{\square\}$ 
```

```
definition  $G \::\ \text{nat}$  where
   $G \equiv \text{max } G1\ G2$ 
```

```
lemma G1:  $G1 \leq G$  and G2:  $G2 \leq G$ 
   $\langle \text{proof} \rangle$ 
```

```
lemma k-ge:  $k1 \geq 2\ k2 \geq 2$ 
   $\langle \text{proof} \rangle$ 
```

```
lemma tm1-tm: turing-machine  $(k1 + k2)\ G\ \text{tm1}$ 
   $\langle \text{proof} \rangle$ 
```

```
lemma tm2-tm: turing-machine  $(k1 + k2)\ G\ \text{tm2}$ 
   $\langle \text{proof} \rangle$ 
```

```
lemma tm3-tm: turing-machine  $(k1 + k2)\ G\ \text{tm3}$ 
   $\langle \text{proof} \rangle$ 
```

```
lemma tm4-tm: turing-machine  $(k1 + k2)\ G\ \text{tm4}$ 
   $\langle \text{proof} \rangle$ 
```

```
lemma tm5-tm: turing-machine  $(k1 + k2)\ G\ \text{tm5}$ 
   $\langle \text{proof} \rangle$ 
```

```
lemma tm6-tm: turing-machine  $(k1 + k2)\ G\ \text{tm6}$ 
   $\langle \text{proof} \rangle$ 
```

```
lemma tm7-tm: turing-machine  $(k1 + k2)\ G\ \text{tm7}$ 
   $\langle \text{proof} \rangle$ 
```

**lemma** *tm8-tm: turing-machine*  $(k1 + k2) G tm8$   
*<proof>*

**context**

**fixes**  $x :: string$

**begin**

**definition**  $zs \equiv string\text{-to-symbols } x$

**lemma** *bit-symbols-zs: bit-symbols*  $zs$   
*<proof>*

**abbreviation**  $n \equiv length\ x$

**lemma** *length-zs [simp]: length*  $zs = n$   
*<proof>*

**definition**  $tps0 \equiv snd\ (start\text{-config } (k1 + k2)\ zs)$

**definition**  $tps1a :: tape\ list$  **where**

$tps1a \equiv SOME\ tps.\ tps :: 1 = string\text{-to-contents } (f1\ x) \wedge$   
 $transforms\ M1\ (snd\ (start\text{-config } k1\ (string\text{-to-symbols } x)))\ (T1\ n)\ tps$

**lemma** *tps1a-aux:*

$tps1a :: 1 = string\text{-to-contents } (f1\ x)$   
 $transforms\ M1\ (snd\ (start\text{-config } k1\ (string\text{-to-symbols } x)))\ (T1\ n)\ tps1a$   
*<proof>*

**lemma** *tps1a:*

$tps1a :: 1 = string\text{-to-contents } (f1\ x)$   
 $transforms\ M1\ (snd\ (start\text{-config } k1\ zs))\ (T1\ n)\ tps1a$   
*<proof>*

**lemma** *length-tps1a [simp]: length*  $tps1a = k1$   
*<proof>*

**definition**  $tps1b :: tape\ list$  **where**

$tps1b \equiv replicate\ k2\ ([\square], 0)$

**definition**  $tps1 :: tape\ list$  **where**

$tps1 \equiv tps1a @ tps1b$

**lemma** *tps1-at-1: tps1 ! 1 = tps1a ! 1*  
*<proof>*

**lemma** *tps1-at-1': tps1 :: 1 = string-to-contents*  $(f1\ x)$   
*<proof>*

**lemma** *tps1-pos-le: tps1 :#: 1 ≤ T1 n*  
*<proof>*

**lemma** *length-f1-x: length*  $(f1\ x) \leq T1\ n$   
*<proof>*

**lemma** *start-config-append:*

$start\text{-config } (k1 + k2)\ zs = (0, snd\ (start\text{-config } k1\ zs) @ tps1b)$   
*<proof>*

**lemma** *tm1 [transforms-intros]: transforms*  $tm1\ tps0\ (T1\ n)\ tps1$   
*<proof>*

**lemma** *clean-string-to-contents: clean-tape*  $(string\text{-to-contents } xs, i)$

*<proof>*

**definition** *tps2* :: *tape list* **where**  
*tps2*  $\equiv$  *tps1* [1 := *tps1* ! 1 |#|= 0]

**lemma** *length-tps2* [*simp*]: *length tps2* = *k1* + *k2*  
*<proof>*

**lemma** *tm2*:  
**assumes** *t* = *Suc* (*T1 n* + *tps1* :#: *Suc 0*)  
**shows** *transforms tm2 tps0 t tps2*  
*<proof>*

**corollary** *tm2'* [*transforms-intros*]:  
**assumes** *t* = *Suc* (*2 \* T1 n*)  
**shows** *transforms tm2 tps0 t tps2*  
*<proof>*

**definition** *tps3* :: *tape list* **where**  
*tps3*  $\equiv$  *tps2* [1 := *tps2* ! 1 |#|= (*Suc* (*length* (*f1 x*))), *k1* := *tps2* ! 1 |#|= (*Suc* (*length* (*f1 x*)))]

**lemma** *tm3*:  
**assumes** *t* = *Suc* (*Suc* (*2 \* T1 n* + *Suc* (*length* (*f1 x*))))  
**shows** *transforms tm3 tps0 t tps3*  
*<proof>*

**definition** *tps3'*  $\equiv$  *tps1a*  
[1 := (*string-to-contents* (*f1 x*), *Suc* (*length* (*f1 x*)))] @  
(*string-to-contents* (*f1 x*), *Suc* (*length* (*f1 x*))) #  
*replicate* (*k2* - 1) ([[]], 0)

**lemma** *tps3'*: *tps3* = *tps3'*  
*<proof>*

**lemma** *tm3'* [*transforms-intros*]:  
**assumes** *t* = *Suc* (*Suc* (*Suc* (*3 \* T1 n*)))  
**shows** *transforms tm3 tps0 t tps3'*  
*<proof>*

**definition** *tps4*  $\equiv$   
*tps1a* [1 := ([[]], 1)] @  
(*string-to-contents* (*f1 x*), *Suc* (*length* (*f1 x*))) #  
*replicate* (*k2* - 1) ([[]], 0)

**lemma** *tm4*:  
**assumes** *t* = *9* + (*3 \* T1 n* + (*Suc* (*3 \* length* (*string-to-symbols* (*f1 x*)))))  
**shows** *transforms tm4 tps0 t tps4*  
*<proof>*

**lemma** *tm4'* [*transforms-intros*]:  
**assumes** *t* = *10* + (*6 \* T1 n*)  
**shows** *transforms tm4 tps0 t tps4*  
*<proof>*

**definition** *tps5*  $\equiv$   
*tps1a* [1 := ([[]], 1)] @  
(*string-to-contents* (*f1 x*), 0) #  
*replicate* (*k2* - 1) ([[]], 0)

**lemma** *tm5*:  
**assumes** *t* = *11* + (*6 \* T1 n* + *tps4* :#: *k1*)  
**shows** *transforms tm5 tps0 t tps5*  
*<proof>*

**abbreviation**  $ys \equiv \text{string-to-symbols } (f1 \ x)$

**abbreviation**  $m \equiv \text{length } (f1 \ x)$

**definition**  $tps5' \equiv$   
 $tps1a [1 := (\llbracket \ \ \rrbracket, 1)] @$   
 $snd (\text{start-config } k2 \ ys)$

**lemma**  $tps5': tps5 = tps5'$   
 $\langle \text{proof} \rangle$

**lemma**  $tm5' [\text{transforms-intros}]$ :  
**assumes**  $t = 12 + 7 * T1 \ n$   
**shows**  $\text{transforms } tm5 \ tps0 \ t \ tps5'$   
 $\langle \text{proof} \rangle$

**definition**  $tps6b :: \text{tape list where}$   
 $tps6b \equiv \text{SOME } tps. tps ::: 1 = \text{string-to-contents } (f2 \ (f1 \ x)) \wedge$   
 $\text{transforms } M2 \ (snd \ (\text{start-config } k2 \ ys)) \ (T2 \ m) \ tps$

**lemma**  $tps6b$ :  
 $tps6b ::: 1 = \text{string-to-contents } (f2 \ (f1 \ x))$   
 $\text{transforms } M2 \ (snd \ (\text{start-config } k2 \ ys)) \ (T2 \ m) \ tps6b$   
 $\langle \text{proof} \rangle$

**lemma**  $tps6b\text{-pos-le}$ :  $tps6b \text{ :\#} : 1 \leq T2 \ m$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{length-tps6b}$ :  $\text{length } tps6b = k2$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{length-f2-f1-x}$ :  $\text{length } (f2 \ (f1 \ x)) \leq T2 \ m$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{enlarged-M2}$ :  $\text{transforms } (\text{enlarged } G2 \ M2) \ (snd \ (\text{start-config } k2 \ ys)) \ (T2 \ m) \ tps6b$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{enlarged-M2-tm}$ :  $\text{turing-machine } k2 \ G \ (\text{enlarged } G2 \ M2)$   
 $\langle \text{proof} \rangle$

**definition**  $tps6 \equiv tps1a[1 := (\llbracket \ \ \rrbracket, 1)] @ tps6b$

**lemma**  $tm56 [\text{transforms-intros}]$ :  $\text{transforms } tm56 \ tps5' \ (T2 \ m) \ tps6$   
 $\langle \text{proof} \rangle$

**lemma**  $tps6\text{-at-Suc-k1}$ :  $tps6 ::: (k1 + 1) = \text{string-to-contents } (f2 \ (f1 \ x))$   
**and**  $tps6\text{-pos-le}$ :  $tps6 \text{ :\#} : (k1 + 1) \leq T2 \ m$   
 $\langle \text{proof} \rangle$

**lemma**  $tm6 [\text{transforms-intros}]$ :  
**assumes**  $t = 12 + 7 * T1 \ n + T2 \ m$   
**shows**  $\text{transforms } tm6 \ tps0 \ t \ tps6$   
 $\langle \text{proof} \rangle$

**definition**  $tps7 \equiv$   
 $tps1a[1 := (\llbracket \ \ \rrbracket, 1)] @$   
 $tps6b[1 := (\text{string-to-contents } (f2 \ (f1 \ x)), 1)]$

**lemma**  $tps7\text{-at-Suc-k1}$ :  $tps7 ! (k1 + 1) = (\text{string-to-contents } (f2 \ (f1 \ x)), 1)$   
 $\langle \text{proof} \rangle$

**lemma**  $tm7$ :

```

assumes  $t = 14 + (7 * T1\ n + (T2\ m + tps6\ :\# : Suc\ k1))$ 
shows transforms tm7 tps0 t tps7
⟨proof⟩

```

```

corollary tm7' [transforms-intros]:
assumes  $t = 14 + 7 * T1\ n + 2 * T2\ m$ 
shows transforms tm7 tps0 t tps7
⟨proof⟩

```

```

definition tps8 ≡
  tps1a[1 := (string-to-contents ( $f2\ (f1\ x)$ ), Suc (length ( $f2\ (f1\ x)$ )))] @
  tps6b[1 := (string-to-contents ( $f2\ (f1\ x)$ ), Suc (length ( $f2\ (f1\ x)$ )))]

```

```

lemma tps8-at-1: tps8 ::  $1 = \text{string-to-contents } (f2\ (f1\ x))$ 
⟨proof⟩

```

```

lemma tm8:
assumes  $t = 15 + 7 * T1\ n + 2 * T2\ m + \text{length } (f2\ (f1\ x))$ 
shows transforms tm8 tps0 t tps8
⟨proof⟩

```

```

lemma tm8':
assumes  $t = 15 + 7 * T1\ n + 3 * T2\ m$ 
shows transforms tm8 tps0 t tps8
⟨proof⟩

```

```

lemma tm8'-mono:
assumes mono T2
and  $t = 15 + 7 * T1\ n + 3 * T2\ (T1\ n)$ 
shows transforms tm8 tps0 t tps8
⟨proof⟩

```

**end**

```

lemma computes-composed-mono:
assumes mono T2 and  $T = (\lambda n. 15 + 7 * T1\ n + 3 * T2\ (T1\ n))$ 
shows computes-in-time ( $k1 + k2$ ) tm8 ( $f2 \circ f1$ )  $T$ 
⟨proof⟩

```

**end**

```

lemma computes-composed-poly:
assumes tm-M1: turing-machine  $k1\ G1\ M1$ 
and tm-M2: turing-machine  $k2\ G2\ M2$ 
and computes1: computes-in-time  $k1\ M1\ f1\ T1$ 
and computes2: computes-in-time  $k2\ M2\ f2\ T2$ 
assumes big-oh-poly T1 and big-oh-poly T2
shows  $\exists T\ k\ G\ M. \text{big-oh-poly } T \wedge \text{turing-machine } k\ G\ M \wedge \text{computes-in-time } k\ M\ (f2 \circ f1)\ T$ 
⟨proof⟩

```

**end**

## 2.7 Arithmetic

```

theory Arithmetic
imports Memorizing
begin

```

In this section we define a representation of natural numbers and some reusable Turing machines for elementary arithmetical operations. All Turing machines implementing the operations assume that the tape heads on the tapes containing the operands and the result(s) contain one natural number each. In programming language terms we could say that such a tape corresponds to a variable of type *nat*. Furthermore, initially the tape heads are on cell number 1, that is, one to the right of the start symbol.

The Turing machines will halt with the tape heads in that position as well. In that way operations can be concatenated seamlessly.

### 2.7.1 Binary numbers

We represent binary numbers as sequences of the symbols **0** and **1**. Slightly unusually the least significant bit will be on the left. While every sequence over these symbols represents a natural number, the representation is not unique due to leading (or rather, trailing) zeros. The *canonical* representation is unique and has no trailing zeros, not even for the number zero, which is thus represented by the empty symbol sequence. As a side effect empty tapes can be thought of as being initialized with zero.

Naturally the binary digits 0 and 1 are represented by the symbols **0** and **1**, respectively. For example, the decimal number 14, conventionally written  $1100_2$  in binary, is represented by the symbol sequence **0011**. The next two functions map between symbols and binary digits:

**abbreviation** (*input*) *tosym* :: *nat*  $\Rightarrow$  *symbol* **where**  
*tosym* *z*  $\equiv$  *z* + 2

**abbreviation** *todigit* :: *symbol*  $\Rightarrow$  *nat* **where**  
*todigit* *z*  $\equiv$  if *z* = **1** then 1 else 0

The numerical value of a symbol sequence:

**definition** *num* :: *symbol list*  $\Rightarrow$  *nat* **where**  
*num* *xs*  $\equiv$   $\sum_{i \leftarrow [0..<length\ xs]}$  *todigit* (*xs* ! *i*) \* 2  $^$  *i*

The *i*-th digit of a symbol sequence, where digits out of bounds are considered trailing zeros:

**definition** *digit* :: *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**  
*digit* *xs* *i*  $\equiv$  if *i* < *length* *xs* then *xs* ! *i* else 0

Some properties of *num*:

**lemma** *num-ge-pow*:  
**assumes** *i* < *length* *xs* **and** *xs* ! *i* = **1**  
**shows** *num* *xs*  $\geq$  2  $^$  *i*  
 <proof>

**lemma** *num-trailing-zero*:  
**assumes** *todigit* *z* = 0  
**shows** *num* *xs* = *num* (*xs* @ [*z*])  
 <proof>

**lemma** *num-Cons*: *num* (*x* # *xs*) = *todigit* *x* + 2 \* *num* *xs*  
 <proof>

**lemma** *num-append*: *num* (*xs* @ *ys*) = *num* *xs* + 2  $^$  *length* *xs* \* *num* *ys*  
 <proof>

**lemma** *num-drop*: *num* (*drop* *t* *zs*) = *todigit* (*digit* *zs* *t*) + 2 \* *num* (*drop* (*Suc* *t*) *zs*)  
 <proof>

**lemma** *num-take-Suc*: *num* (*take* (*Suc* *t*) *zs*) = *num* (*take* *t* *zs*) + 2  $^$  *t* \* *todigit* (*digit* *zs* *t*)  
 <proof>

A symbol sequence is a canonical representation of a natural number if the sequence contains only the symbols **0** and **1** and is either empty or ends in **1**.

**definition** *canonical* :: *symbol list*  $\Rightarrow$  *bool* **where**  
*canonical* *xs*  $\equiv$  *bit-symbols* *xs*  $\wedge$  (*xs* = []  $\vee$  *last* *xs* = **1**)

**lemma** *canonical-Cons*:  
**assumes** *canonical* *xs* **and** *xs*  $\neq$  [] **and** *x* = **0**  $\vee$  *x* = **1**  
**shows** *canonical* (*x* # *xs*)  
 <proof>

**lemma canonical-Cons-3:** *canonical xs*  $\implies$  *canonical (1 # xs)*  
(proof)

**lemma canonical-tl:** *canonical (x # xs)*  $\implies$  *canonical xs*  
(proof)

**lemma prepend-2-even:**  $x = \mathbf{0} \implies \text{even (num (x # xs))}$   
(proof)

**lemma prepend-3-odd:**  $x = \mathbf{1} \implies \text{odd (num (x # xs))}$   
(proof)

Every number has exactly one canonical representation.

**lemma canonical-ex1:**  
fixes  $n :: \text{nat}$   
shows  $\exists! xs. \text{num } xs = n \wedge \text{canonical } xs$   
(proof)

The canonical representation of a natural number as symbol sequence:

**definition canrepr**  $:: \text{nat} \Rightarrow \text{symbol list}$  **where**  
 $\text{canrepr } n \equiv \text{THE } xs. \text{num } xs = n \wedge \text{canonical } xs$

**lemma canrepr-inj:** *inj canrepr*  
(proof)

**lemma canonical-canrepr:** *canonical (canrepr n)*  
(proof)

**lemma canrepr:**  $\text{num (canrepr } n) = n$   
(proof)

**lemma bit-symbols-canrepr:** *bit-symbols (canrepr n)*  
(proof)

**lemma proper-symbols-canrepr:** *proper-symbols (canrepr n)*  
(proof)

**lemma canreprI:**  $\text{num } xs = n \implies \text{canonical } xs \implies \text{canrepr } n = xs$   
(proof)

**lemma canrepr-0:**  $\text{canrepr } 0 = []$   
(proof)

**lemma canrepr-1:**  $\text{canrepr } 1 = [1]$   
(proof)

The length of the canonical representation of a number  $n$ :

**abbreviation nlength**  $:: \text{nat} \Rightarrow \text{nat}$  **where**  
 $\text{nlength } n \equiv \text{length (canrepr } n)$

**lemma nlength-0:**  $\text{nlength } n = 0 \iff n = 0$   
(proof)

**corollary nlength-0-simp [simp]:**  $\text{nlength } 0 = 0$   
(proof)

**lemma num-replicate2-eq-pow:**  $\text{num (replicate } j \mathbf{0} @ [1]) = 2^j$   
(proof)

**lemma num-replicate3-eq-pow-minus-1:**  $\text{num (replicate } j \mathbf{1}) = 2^j - 1$   
(proof)

**lemma nlength-pow2:**  $\text{nlength } (2^j) = \text{Suc } j$

*<proof>*

**corollary** *nlength-1-simp* [*simp*]:  $nlength\ 1 = 1$   
*<proof>*

**corollary** *nlength-2*:  $nlength\ 2 = 2$   
*<proof>*

**lemma** *nlength-pow-minus-1*:  $nlength\ (2^j - 1) = j$   
*<proof>*

**corollary** *nlength-3*:  $nlength\ 3 = 2$   
*<proof>*

When handling natural numbers, Turing machines will usually have tape contents of the following form:

**abbreviation** *ncontents* ::  $nat \Rightarrow (nat \Rightarrow symbol)$  ( $\langle \_ \rangle_N$ ) **where**  
 $\lfloor n \rfloor_N \equiv \lfloor canrepr\ n \rfloor$

**lemma** *ncontents-0*:  $\lfloor 0 \rfloor_N = \lfloor [] \rfloor$   
*<proof>*

**lemma** *clean-tape-ncontents*:  $clean-tape\ (\lfloor x \rfloor_N, i)$   
*<proof>*

**lemma** *ncontents-1-blank-iff-zero*:  $\lfloor n \rfloor_N\ 1 = \square \iff n = 0$   
*<proof>*

Every bit symbol sequence can be turned into a canonical representation of some number by stripping trailing zeros. The length of the prefix without trailing zeros is given by the next function:

**definition** *canlen* ::  $symbol\ list \Rightarrow nat$  **where**  
 $canlen\ zs \equiv LEAST\ m. \forall i < length\ zs. i \geq m \longrightarrow zs\ !\ i = 0$

**lemma** *canlen-at-ge*:  $\forall i < length\ zs. i \geq canlen\ zs \longrightarrow zs\ !\ i = 0$   
*<proof>*

**lemma** *canlen-eqI*:  
**assumes**  $\forall i < length\ zs. i \geq m \longrightarrow zs\ !\ i = 0$   
**and**  $\bigwedge y. \forall i < length\ zs. i \geq y \longrightarrow zs\ !\ i = 0 \implies m \leq y$   
**shows**  $canlen\ zs = m$   
*<proof>*

**lemma** *canlen-le-length*:  $canlen\ zs \leq length\ zs$   
*<proof>*

**lemma** *canlen-le*:  
**assumes**  $\forall i < length\ zs. i \geq m \longrightarrow zs\ !\ i = 0$   
**shows**  $m \geq canlen\ zs$   
*<proof>*

**lemma** *canlen-one*:  
**assumes** *bit-symbols* *zs* **and**  $canlen\ zs > 0$   
**shows**  $zs\ !\ (canlen\ zs - 1) = 1$   
*<proof>*

**lemma** *canonical-take-canlen*:  
**assumes** *bit-symbols* *zs*  
**shows**  $canonical\ (take\ (canlen\ zs)\ zs)$   
*<proof>*

**lemma** *num-take-canlen-eq*:  $num\ (take\ (canlen\ zs)\ zs) = num\ zs$   
*<proof>*

**lemma** *canrepr-take-canlen*:

**assumes**  $num\ zs = n$  **and**  $bit\text{-}symbols\ zs$   
**shows**  $canrepr\ n = take\ (canlen\ zs)\ zs$   
 $\langle proof \rangle$

**lemma**  $length\text{-}canrepr\text{-}canlen$ :  
**assumes**  $num\ zs = n$  **and**  $bit\text{-}symbols\ zs$   
**shows**  $nlength\ n = canlen\ zs$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}ge\text{-}pow$ :  
**assumes**  $nlength\ n = Suc\ j$   
**shows**  $n \geq 2^j$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}less\text{-}pow$ :  $n < 2^{(nlength\ n)}$   
 $\langle proof \rangle$

**lemma**  $pow\text{-}nlength$ :  
**assumes**  $2^j \leq n$  **and**  $n < 2^{(Suc\ j)}$   
**shows**  $nlength\ n = Suc\ j$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}le\text{-}n$ :  $nlength\ n \leq n$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}Suc\text{-}le$ :  $nlength\ n \leq nlength\ (Suc\ n)$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}mono$ :  
**assumes**  $n1 \leq n2$   
**shows**  $nlength\ n1 \leq nlength\ n2$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}even\text{-}le$ :  $n > 0 \implies nlength\ (2 * n) = Suc\ (nlength\ n)$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}prod$ :  $nlength\ (n1 * n2) \leq nlength\ n1 + nlength\ n2$   
 $\langle proof \rangle$

In the following lemma  $Suc$  is needed because  $n^0 = 1$ .

**lemma**  $nlength\text{-}pow$ :  $nlength\ (n^d) \leq Suc\ (d * nlength\ n)$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}sum$ :  $nlength\ (n1 + n2) \leq Suc\ (max\ (nlength\ n1)\ (nlength\ n2))$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}Suc$ :  $nlength\ (Suc\ n) \leq Suc\ (nlength\ n)$   
 $\langle proof \rangle$

**lemma**  $nlength\text{-}less\text{-}n$ :  $n \geq 3 \implies nlength\ n < n$   
 $\langle proof \rangle$

## Comparing two numbers

In order to compare two numbers in canonical representation, we can use the Turing machine  $tm\text{-}equals$ , which works for arbitrary proper symbol sequences.

**lemma**  $min\text{-}nlength$ :  $min\ (nlength\ n1)\ (nlength\ n2) = nlength\ (min\ n1\ n2)$   
 $\langle proof \rangle$

**lemma**  $max\text{-}nlength$ :  $max\ (nlength\ n1)\ (nlength\ n2) = nlength\ (max\ n1\ n2)$   
 $\langle proof \rangle$

**lemma** *contents-blank-0*:  $\lfloor [\square] \rfloor = \lfloor [] \rfloor$   
 ⟨*proof*⟩

**definition** *tm-equalsn* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**  
*tm-equalsn*  $\equiv$  *tm-equals*

**lemma** *tm-equalsn-tm*:  
**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j3$   
**shows** *turing-machine*  $k$   $G$  (*tm-equalsn*  $j1$   $j2$   $j3$ )  
 ⟨*proof*⟩

**lemma** *transforms-tm-equalsnI* [*transforms-intros*]:  
**fixes**  $j1$   $j2$   $j3$  :: *tapeidx*  
**fixes**  $tps$   $tps'$  :: *tape list* **and**  $k$   $b$   $n1$   $n2$  :: *nat*  
**assumes**  $\text{length } tps = k$   $j1 \neq j2$   $j2 \neq j3$   $j1 \neq j3$   $j1 < k$   $j2 < k$   $j3 < k$   
**and**  $b \leq 1$   
**assumes**  
 $tps ! j1 = (\lfloor n1 \rfloor_N, 1)$   
 $tps ! j2 = (\lfloor n2 \rfloor_N, 1)$   
 $tps ! j3 = (\lfloor b \rfloor_N, 1)$   
**assumes**  $tts = (3 * \text{nlength } (\text{min } n1 \ n2) + 7)$   
**assumes**  $tps' = tps$   
 $[j3 := (\lfloor \text{if } n1 = n2 \text{ then } 1 \text{ else } 0 \rfloor_N, 1)]$   
**shows** *transforms* (*tm-equalsn*  $j1$   $j2$   $j3$ )  $tps$   $tts$   $tps'$   
 ⟨*proof*⟩

## Copying a number between tapes

The next Turing machine overwrites the contents of tape  $j_2$  with the contents of tape  $j_1$  and performs a carriage return on both tapes.

**definition** *tm-copyn* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**  
*tm-copyn*  $j1$   $j2$   $\equiv$   
*tm-erase-cr*  $j2$  ;;  
*tm-cp-until*  $j1$   $j2$   $\{\square\}$  ;;  
*tm-cr*  $j1$  ;;  
*tm-cr*  $j2$

**lemma** *tm-copyn-tm*:  
**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $j1 < k$   $j2 < k$   $j1 \neq j2$   $0 < j2$   
**shows** *turing-machine*  $k$   $G$  (*tm-copyn*  $j1$   $j2$ )  
 ⟨*proof*⟩

**locale** *turing-machine-move* =  
**fixes**  $j1$   $j2$  :: *tapeidx*  
**begin**

**definition**  $tm1 \equiv \text{tm-erase-cr } j2$   
**definition**  $tm2 \equiv tm1$  ;; *tm-cp-until*  $j1$   $j2$   $\{\square\}$   
**definition**  $tm3 \equiv tm2$  ;; *tm-cr*  $j1$   
**definition**  $tm4 \equiv tm3$  ;; *tm-cr*  $j2$

**lemma** *tm4-eq-tm-copyn*:  $tm4 = \text{tm-copyn } j1$   $j2$   
 ⟨*proof*⟩

**context**  
**fixes**  $x$   $y$  :: *nat* **and**  $tps0$  :: *tape list*  
**assumes** *j-less* [*simp*]:  $j1 < \text{length } tps0$   $j2 < \text{length } tps0$   
**assumes**  $j$  [*simp*]:  $j1 \neq j2$   
**and**  $tps-j1$  [*simp*]:  $tps0 ! j1 = (\lfloor x \rfloor_N, 1)$   
**and**  $tps-j2$  [*simp*]:  $tps0 ! j2 = (\lfloor y \rfloor_N, 1)$   
**begin**

**definition**  $tps1 \equiv tps0$

$[j2 := ([\ ], 1)]$

**lemma** *tm1* [*transforms-intros*]:  
 **assumes**  $t = 7 + 2 * \text{nlength } y$   
 **shows** *transforms tm1 tps0 t tps1*  
  $\langle \text{proof} \rangle$

**definition** *tps2*  $\equiv \text{tps0}$   
  $[j1 := ([x]_N, \text{Suc } (\text{nlength } x)),$   
  $j2 := ([x]_N, \text{Suc } (\text{nlength } x))]$

**lemma** *tm2* [*transforms-intros*]:  
 **assumes**  $t = 8 + (2 * \text{nlength } y + \text{nlength } x)$   
 **shows** *transforms tm2 tps0 t tps2*  
  $\langle \text{proof} \rangle$

**definition** *tps3*  $\equiv \text{tps0}[j2 := ([x]_N, \text{Suc } (\text{nlength } x))]$

**lemma** *tm3* [*transforms-intros*]:  
 **assumes**  $t = 11 + (2 * \text{nlength } y + 2 * \text{nlength } x)$   
 **shows** *transforms tm3 tps0 t tps3*  
  $\langle \text{proof} \rangle$

**definition** *tps4*  $\equiv \text{tps0}[j2 := ([x]_N, 1)]$

**lemma** *tm4*:  
 **assumes**  $t = 14 + (3 * \text{nlength } x + 2 * \text{nlength } y)$   
 **shows** *transforms tm4 tps0 t tps4*  
  $\langle \text{proof} \rangle$

**lemma** *tm4'*:  
 **assumes**  $t = 14 + 3 * (\text{nlength } x + \text{nlength } y)$   
 **shows** *transforms tm4 tps0 t tps4*  
  $\langle \text{proof} \rangle$

**end**

**end**

**lemma** *transforms-tm-copynI* [*transforms-intros*]:  
 **fixes**  $j1 j2 :: \text{tapeidx}$   
 **fixes**  $\text{tps } \text{tps}' :: \text{tape list}$  **and**  $k x y :: \text{nat}$   
 **assumes**  $j1 \neq j2$   $j1 < \text{length } \text{tps}$   $j2 < \text{length } \text{tps}$   
 **assumes**  
  $\text{tps } ! j1 = ([x]_N, 1)$   
  $\text{tps } ! j2 = ([y]_N, 1)$   
 **assumes**  $\text{tnt} = 14 + 3 * (\text{nlength } x + \text{nlength } y)$   
 **assumes**  $\text{tps}' = \text{tps}$   
  $[j2 := ([x]_N, 1)]$   
 **shows** *transforms (tm-copyn j1 j2) tps tnt tps'*  
  $\langle \text{proof} \rangle$

## Setting the tape contents to a number

The Turing machine in this section writes a hard-coded number to a tape.

**definition** *tm-setn*  $:: \text{tapeidx} \Rightarrow \text{nat} \Rightarrow \text{machine}$  **where**  
  $\text{tm-setn } j n \equiv \text{tm-set } j (\text{canrepr } n)$

**lemma** *tm-setn-tm*:  
 **assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $j < k$  **and**  $0 < j$   
 **shows** *turing-machine k G (tm-setn j n)*  
  $\langle \text{proof} \rangle$

**lemma** *transforms-tm-setnI* [*transforms-intros*]:  
**fixes**  $j :: \text{tapeidx}$   
**fixes**  $\text{tps tps}' :: \text{tape list}$  **and**  $x k n :: \text{nat}$   
**assumes**  $j < \text{length tps}$   
**assumes**  $\text{tps} ! j = (\lfloor x \rfloor_N, 1)$   
**assumes**  $t = 10 + 2 * \text{nlength } x + 2 * \text{nlength } n$   
**assumes**  $\text{tps}' = \text{tps}[j := (\lfloor n \rfloor_N, 1)]$   
**shows** *transforms (tm-setn j n) tps t tps'*  
*<proof>*

## 2.7.2 Incrementing

In this section we devise a Turing machine that increments a number. The next function describes how the symbol sequence of the incremented number looks like. Basically one has to flip all **1** symbols starting at the least significant digit until one reaches a **0**, which is then replaced by a **1**. If there is no **0**, a **1** is appended. Here we exploit that the most significant digit is to the right.

**definition** *nincr* :: *symbol list*  $\Rightarrow$  *symbol list* **where**  
*nincr*  $zs \equiv$   
 if  $\exists i < \text{length } zs. zs ! i = \mathbf{0}$   
 then *replicate (LEAST i. i < length zs  $\wedge$  zs ! i = 0) 0 @ [1] @ drop (Suc (LEAST i. i < length zs  $\wedge$  zs ! i = 0)) zs*  
 else *replicate (length zs) 0 @ [1]*

**lemma** *canonical-nincr*:  
**assumes** *canonical zs*  
**shows** *canonical (nincr zs)*  
*<proof>*

**lemma** *nincr*:  
**assumes** *bit-symbols zs*  
**shows**  $\text{num (nincr } zs) = \text{Suc (num } zs)$   
*<proof>*

**lemma** *nincr-canrepr*:  $\text{nincr (canrepr } n) = \text{canrepr (Suc } n)$   
*<proof>*

The next Turing machine performs the incrementing. Starting from the left of the symbol sequence on tape  $j$ , it writes the symbol **0** until it reaches a blank or the symbol **1**. Then it writes a **1** and returns the tape head to the beginning.

**definition** *tm-incr* :: *tapeidx*  $\Rightarrow$  *machine* **where**  
*tm-incr j*  $\equiv \text{tm-const-until } j j \{\square, \mathbf{0}\} \mathbf{0} ;; \text{tm-write } j \mathbf{1} ;; \text{tm-cr } j$

**lemma** *tm-incr-tm*:  
**assumes**  $G \geq 4$  **and**  $k \geq 2$  **and**  $j < k$  **and**  $j > 0$   
**shows** *turing-machine k G (tm-incr j)*  
*<proof>*

**locale** *turing-machine-incr* =  
**fixes**  $j :: \text{tapeidx}$   
**begin**

**definition**  $\text{tm1} \equiv \text{tm-const-until } j j \{\square, \mathbf{0}\} \mathbf{0}$

**definition**  $\text{tm2} \equiv \text{tm1} ;; \text{tm-write } j \mathbf{1}$

**definition**  $\text{tm3} \equiv \text{tm2} ;; \text{tm-cr } j$

**lemma** *tm3-eq-tm-incr*:  $\text{tm3} = \text{tm-incr } j$   
*<proof>*

**context**

**fixes**  $x k :: \text{nat}$  **and**  $\text{tps} :: \text{tape list}$   
**assumes**  $\text{jk [simp]: } j < k \text{ length tps} = k$   
**and**  $\text{tps0 [simp]: tps} ! j = (\lfloor x \rfloor_N, 1)$

**begin**

**lemma** *tm1* [*transforms-intros*]:  
 **assumes**  $i0 = (LEAST\ i.\ i \leq nlength\ x \wedge \lfloor x \rfloor_N (Suc\ i) \in \{\square, \mathbf{0}\})$   
 **and**  $tps' = tps[j := constplant\ (tps\ !\ j)\ \mathbf{0}\ i0]$   
 **shows** *transforms tm1 tps (Suc i0) tps'*  
 *<proof>*

**lemma** *tm2* [*transforms-intros*]:  
 **assumes**  $i0 = (LEAST\ i.\ i \leq nlength\ x \wedge \lfloor x \rfloor_N (Suc\ i) \in \{\square, \mathbf{0}\})$   
 **and**  $ttt = Suc\ (Suc\ i0)$   
 **and**  $tps' = tps[j := (\lfloor Suc\ x \rfloor_N, Suc\ i0)]$   
 **shows** *transforms tm2 tps ttt tps'*  
 *<proof>*

**lemma** *tm3*:  
 **assumes**  $i0 = (LEAST\ i.\ i \leq nlength\ x \wedge \lfloor x \rfloor_N (Suc\ i) \in \{\square, \mathbf{0}\})$   
 **and**  $ttt = 5 + 2 * i0$   
 **and**  $tps' = tps[j := (\lfloor Suc\ x \rfloor_N, Suc\ 0)]$   
 **shows** *transforms tm3 tps ttt tps'*  
 *<proof>*

**lemma** *tm3'*:  
 **assumes**  $ttt = 5 + 2 * nlength\ x$   
 **and**  $tps' = tps[j := (\lfloor Suc\ x \rfloor_N, Suc\ 0)]$   
 **shows** *transforms tm3 tps ttt tps'*  
 *<proof>*

**end**

**end**

**lemma** *transforms-tm-incrI* [*transforms-intros*]:  
 **assumes**  $j < k$   
 **and**  $length\ tps = k$   
 **and**  $tps\ !\ j = (\lfloor x \rfloor_N, 1)$   
 **and**  $ttt = 5 + 2 * nlength\ x$   
 **and**  $tps' = tps[j := (\lfloor Suc\ x \rfloor_N, 1)]$   
 **shows** *transforms (tm-incr j) tps ttt tps'*  
 *<proof>*

## Incrementing multiple times

Adding a constant by iteratively incrementing is not exactly efficient, but it still only takes constant time and thus does not endanger any time bounds.

**fun** *tm-plus-const* ::  $nat \Rightarrow tapeidx \Rightarrow machine$  **where**  
 *tm-plus-const* 0  $j = []$  |  
 *tm-plus-const* (Suc  $c$ )  $j = tm-plus-const\ c\ j ;; tm-incr\ j$

**lemma** *tm-plus-const-tm*:  
 **assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j$  **and**  $j < k$   
 **shows** *turing-machine k G (tm-plus-const c j)*  
 *<proof>*

**lemma** *transforms-tm-plus-constI* [*transforms-intros*]:  
 **fixes**  $c :: nat$   
 **assumes**  $j < k$   
 **and**  $j > 0$   
 **and**  $length\ tps = k$   
 **and**  $tps\ !\ j = (\lfloor x \rfloor_N, 1)$   
 **and**  $ttt = c * (5 + 2 * nlength\ (x + c))$   
 **and**  $tps' = tps[j := (\lfloor x + c \rfloor_N, 1)]$   
 **shows** *transforms (tm-plus-const c j) tps ttt tps'*

*<proof>*

### 2.7.3 Decrementing

Decrementing a number is almost like incrementing but with the symbols **0** and **1** swapped. One difference is that in order to get a canonical symbol sequence, a trailing zero must be removed, whereas incrementing cannot result in a trailing zero. Another difference is that decrementing the number zero yields zero.

The next function returns the leftmost symbol **1**, that is, the one that needs to be flipped.

**definition** *first1* :: *symbol list*  $\Rightarrow$  *nat* **where**  
*first1* *zs*  $\equiv$  *LEAST* *i*. *i* < *length* *zs*  $\wedge$  *zs* ! *i* = **1**

**lemma** *canonical-ex-3*:  
**assumes** *canonical* *zs* **and** *zs*  $\neq$  []  
**shows**  $\exists i < \text{length } zs. zs ! i = \mathbf{1}$   
*<proof>*

**lemma** *canonical-first1*:  
**assumes** *canonical* *zs* **and** *zs*  $\neq$  []  
**shows** *first1* *zs* < *length* *zs*  $\wedge$  *zs* ! *first1* *zs* = **1**  
*<proof>*

**lemma** *canonical-first1-less*:  
**assumes** *canonical* *zs* **and** *zs*  $\neq$  []  
**shows**  $\forall i < \text{first1 } zs. zs ! i = \mathbf{0}$   
*<proof>*

The next function describes how the canonical representation of the decremented symbol sequence looks like. It has special cases for the empty sequence and for sequences whose only **1** is the most significant digit.

**definition** *ndecr* :: *symbol list*  $\Rightarrow$  *symbol list* **where**  
*ndecr* *zs*  $\equiv$   
  if *zs* = [] then []  
  else if *first1* *zs* = *length* *zs* - 1  
    then replicate (*first1* *zs*) **1**  
    else replicate (*first1* *zs*) **1** @ [0] @ drop (*Suc* (*first1* *zs*)) *zs*

**lemma** *canonical-ndecr*:  
**assumes** *canonical* *zs*  
**shows** *canonical* (*ndecr* *zs*)  
*<proof>*

**lemma** *ndecr*:  
**assumes** *canonical* *zs*  
**shows** *num* (*ndecr* *zs*) = *num* *zs* - 1  
*<proof>*

The next Turing machine implements the function *ndecr*. It does nothing on the empty input, which represents zero. On other inputs it writes symbols **1** going right until it reaches a **1** symbol, which is guaranteed to happen for non-empty canonical representations. It then overwrites this **1** with **0**. If there is a blank symbol to the right of this **0**, the **0** is removed again.

**definition** *tm-decr* :: *tapeidx*  $\Rightarrow$  *machine* **where**  
*tm-decr* *j*  $\equiv$   
  IF  $\lambda rs. rs ! j = \square$  THEN  
    []  
  ELSE  
    *tm-const-until* *j* *j* {**1**} **1** ;;  
    *tm-rtrans* *j* ( $\lambda -. \mathbf{0}$ ) ;;  
    IF  $\lambda rs. rs ! j = \square$  THEN  
      *tm-left* *j* ;;  
      *tm-write* *j*  $\square$   
    ELSE

```

    []
  ENDIF ;;
  tm-cr j
ENDIF

```

**lemma** *tm-decr-tm*:

**assumes**  $G \geq 4$  **and**  $k \geq 2$  **and**  $j < k$  **and**  $0 < j$   
**shows** *turing-machine*  $k$   $G$  (*tm-decr*  $j$ )  
*<proof>*

**locale** *turing-machine-decr* =

**fixes**  $j :: \text{tapeid}$

**begin**

**definition**  $tm1 \equiv \text{tm-const-until } j \ j \ \{1\} \ 1$

**definition**  $tm2 \equiv tm1 \ ; \ ; \ \text{tm-rtrans } j \ (\lambda-. \ 0)$

**definition**  $tm23 \equiv \text{tm-left } j$

**definition**  $tm24 \equiv tm23 \ ; \ ; \ \text{tm-write } j \ \square$

**definition**  $tm25 \equiv \text{IF } \lambda rs. rs \ ! \ j = \square \ \text{THEN } tm24 \ \text{ELSE } [] \ \text{ENDIF}$

**definition**  $tm5 \equiv tm2 \ ; \ ; \ tm25$

**definition**  $tm6 \equiv tm5 \ ; \ ; \ \text{tm-cr } j$

**definition**  $tm7 \equiv \text{IF } \lambda rs. rs \ ! \ j = \square \ \text{THEN } [] \ \text{ELSE } tm6 \ \text{ENDIF}$

**lemma** *tm7-eq-tm-decr*:  $tm7 = \text{tm-decr } j$

*<proof>*

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $xs :: \text{symbol list}$  **and**  $k :: \text{nat}$

**assumes**  $jk: \text{length } tps0 = k \ j < k$

**and**  $can: \text{canonical } xs$

**and**  $tps0: tps0 \ ! \ j = (\lfloor xs \rfloor, 1)$

**begin**

**lemma** *bs: bit-symbols*  $xs$

*<proof>*

**context**

**assumes**  $read-tps0: \text{read } tps0 \ ! \ j = \square$

**begin**

**lemma** *xs-Nil*:  $xs = []$

*<proof>*

**lemma** *transforms-NilI*:

**assumes**  $ttt = 0$

**and**  $tps' = tps0[j := (\lfloor ndecr \ xs \rfloor, 1)]$

**shows** *transforms*  $[] \ tps0 \ ttt \ tps'$

*<proof>*

**end**

**context**

**assumes**  $read-tps0': \text{read } tps0 \ ! \ j \neq \square$

**begin**

**lemma** *xs: xs  $\neq$  []*

*<proof>*

**lemma** *first1*:  $\text{first1 } xs < \text{length } xs \ xs \ ! \ \text{first1 } xs = 1 \ \forall i < \text{first1 } xs. \ xs \ ! \ i = 0$

*<proof>*

**definition**  $tps1 \equiv tps0$

$[j := (\lfloor \text{replicate } (\text{first1 } xs) \ 1 \ @ \ [1] \ @ \ (\text{drop } (\text{Suc } (\text{first1 } xs)) \ xs)], \ \text{Suc } (\text{first1 } xs))]$

**lemma** *tm1* [*transforms-intros*]:  
**assumes**  $ttt = \text{Suc } (\text{first1 } xs)$   
**shows** *transforms tm1 tps0 ttt tps1*  
 $\langle \text{proof} \rangle$

**definition** *tps2*  $\equiv$  *tps0*  
 $[j := (\llbracket \text{replicate } (\text{first1 } xs) \ \mathbf{1} \ @ \ [\mathbf{0}] \ @ \ \text{drop } (\text{Suc } (\text{first1 } xs)) \ xs \rrbracket, \text{Suc } (\text{Suc } (\text{first1 } xs)))]$

**lemma** *tm2* [*transforms-intros*]:  
**assumes**  $ttt = \text{first1 } xs + 2$   
**shows** *transforms tm2 tps0 ttt tps2*  
 $\langle \text{proof} \rangle$

**definition** *tps5*  $\equiv$  *tps0*  
 $[j := (\llbracket \text{ndecr } xs \rrbracket, \text{if read } tps2 \ ! \ j = \square \ \text{then } \text{Suc } (\text{first1 } xs) \ \text{else } \text{Suc } (\text{Suc } (\text{first1 } xs)))]$

**context**  
**assumes** *read-tps2*: *read tps2 ! j = □*  
**begin**

**lemma** *proper-contents-outofbounds*:  
**assumes** *proper-symbols zs* **and**  $\llbracket zs \rrbracket \ i = \square$   
**shows**  $i > \text{length } zs$   
 $\langle \text{proof} \rangle$

**lemma** *first1-eq*:  $\text{first1 } xs = \text{length } xs - 1$   
 $\langle \text{proof} \rangle$

**lemma** *drop-xs-Nil*:  $\text{drop } (\text{Suc } (\text{first1 } xs)) \ xs = []$   
 $\langle \text{proof} \rangle$

**lemma** *tps2-eq*:  $tps2 = tps0[j := (\llbracket \text{replicate } (\text{first1 } xs) \ \mathbf{1} \ @ \ [\mathbf{0}] \rrbracket, \text{Suc } (\text{Suc } (\text{first1 } xs)))]$   
 $\langle \text{proof} \rangle$

**definition** *tps23*  $\equiv$  *tps0*  
 $[j := (\llbracket \text{replicate } (\text{first1 } xs) \ \mathbf{1} \ @ \ [\mathbf{0}] \rrbracket, \text{Suc } (\text{first1 } xs))]$

**lemma** *tm23* [*transforms-intros*]:  
**assumes**  $ttt = 1$   
**shows** *transforms tm23 tps2 ttt tps23*  
 $\langle \text{proof} \rangle$

**definition** *tps24*  $\equiv$  *tps0*  
 $[j := (\llbracket \text{replicate } (\text{first1 } xs) \ \mathbf{1} \rrbracket, \text{Suc } (\text{first1 } xs))]$

**lemma** *tm24*:  
**assumes**  $ttt = 2$   
**shows** *transforms tm24 tps2 ttt tps24*  
 $\langle \text{proof} \rangle$

**corollary** *tm24'* [*transforms-intros*]:  
**assumes**  $ttt = 2$  **and**  $tps' = tps0[j := (\llbracket \text{ndecr } xs \rrbracket, \text{Suc } (\text{first1 } xs))]$   
**shows** *transforms tm24 tps2 ttt tps'*  
 $\langle \text{proof} \rangle$

**end**

**context**  
**assumes** *read-tps2'*: *read tps2 ! j ≠ □*  
**begin**

**lemma** *first1-neq*:  $\text{first1 } xs \neq \text{length } xs - 1$

*<proof>*

**lemma** *tps2*:  $tps2 = tps0[j := (\lfloor ndecr\ xs \rfloor, Suc\ (Suc\ (first1\ xs)))]$   
*<proof>*

**end**

**lemma** *tm25* [*transforms-intros*]:  
**assumes**  $ttt = (if\ read\ tps2\ !\ j = \square\ then\ 4\ else\ 1)$   
**shows** *transforms tm25 tps2 ttt tps5*  
*<proof>*

**lemma** *tm5* [*transforms-intros*]:  
**assumes**  $ttt = first1\ xs + 2 + (if\ read\ tps2\ !\ j = \square\ then\ 4\ else\ 1)$   
**shows** *transforms tm5 tps0 ttt tps5*  
*<proof>*

**definition** *tps6*  $\equiv tps0$   
 $[j := (\lfloor ndecr\ xs \rfloor, 1)]$

**lemma** *tm6*:  
**assumes**  $ttt = first1\ xs + 2 + (if\ read\ tps2\ !\ j = \square\ then\ 4\ else\ 1) + (tps5\ :\#: j + 2)$   
**shows** *transforms tm6 tps0 ttt tps6*  
*<proof>*

**lemma** *tm6'* [*transforms-intros*]:  
**assumes**  $ttt = 2 * first1\ xs + 9$   
**shows** *transforms tm6 tps0 ttt tps6*  
*<proof>*

**end**

**definition** *tps7*  $\equiv tps0[j := (\lfloor ndecr\ xs \rfloor, 1)]$

**lemma** *tm7*:  
**assumes**  $ttt = 8 + 2 * length\ xs$   
**shows** *transforms tm7 tps0 ttt tps7*  
*<proof>*

**end**

**end**

**lemma** *transforms-tm-decrI* [*transforms-intros*]:  
**fixes** *tps tps'* :: *tape list* **and** *n* :: *nat* **and** *k ttt* :: *nat*  
**assumes**  $j < k\ length\ tps = k$   
**assumes**  $tps\ !\ j = (\lfloor n \rfloor_N, 1)$   
**assumes**  $ttt = 8 + 2 * nlength\ n$   
**assumes**  $tps' = tps[j := (\lfloor n - 1 \rfloor_N, 1)]$   
**shows** *transforms (tm-decr j) tps ttt tps'*  
*<proof>*

## 2.7.4 Addition

In this section we construct a Turing machine that adds two numbers in canonical representation each given on a separate tape and overwrites the second number with the sum. The TM implements the common algorithm with carry starting from the least significant digit.

Given two symbol sequences *xs* and *ys* representing numbers, the next function computes the carry bit that occurs in the *i*-th position. For the least significant position, 0, there is no carry (that is, it is 0); for position *i* + 1 the carry is the sum of the bits of *xs* and *ys* in position *i* and the carry for position *i*. The function gives the carry as symbol **0** or **1**, except for position 0, where it is the start symbol  $\triangleright$ . The start symbol represents the same bit as the symbol **0** as defined by *todigit*. The reason for this special

treatment is that the TM will store the carry on a memorization tape (see Section 2.5), which initially contains the start symbol.

**fun** *carry* :: *symbol list*  $\Rightarrow$  *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *symbol* **where**  
*carry* *xs* *ys* 0 = 1 |  
*carry* *xs* *ys* (Suc *i*) = *tosym* ((*todigit* (*digit* *xs* *i*) + *todigit* (*digit* *ys* *i*) + *todigit* (*carry* *xs* *ys* *i*)) div 2)

The next function specifies the *i*-th digit of the sum.

**definition** *sumdigit* :: *symbol list*  $\Rightarrow$  *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *symbol* **where**  
*sumdigit* *xs* *ys* *i*  $\equiv$  *tosym* ((*todigit* (*digit* *xs* *i*) + *todigit* (*digit* *ys* *i*) + *todigit* (*carry* *xs* *ys* *i*)) mod 2)

**lemma** *carry-sumdigit*: *todigit* (*sumdigit* *xs* *ys* *i*) + 2 \* (*todigit* (*carry* *xs* *ys* (Suc *i*))) =  
*todigit* (*carry* *xs* *ys* *i*) + *todigit* (*digit* *xs* *i*) + *todigit* (*digit* *ys* *i*)  
<proof>

**lemma** *carry-sumdigit-eq-sum*:  
*num* *xs* + *num* *ys* =  
*num* (*map* (*sumdigit* *xs* *ys*) [0..*t*]) + 2 ^ *t* \* *todigit* (*carry* *xs* *ys* *t*) + 2 ^ *t* \* *num* (*drop* *t* *xs*) + 2 ^ *t* \* *num* (*drop* *t* *ys*)  
<proof>

**lemma** *carry-le*:  
**assumes** *symbols-lt* 4 *xs* **and** *symbols-lt* 4 *ys*  
**shows** *carry* *xs* *ys* *t*  $\leq$  1  
<proof>

**lemma** *num-sumdigit-eq-sum*:  
**assumes** *length* *xs*  $\leq$  *n*  
**and** *length* *ys*  $\leq$  *n*  
**and** *symbols-lt* 4 *xs*  
**and** *symbols-lt* 4 *ys*  
**shows** *num* *xs* + *num* *ys* = *num* (*map* (*sumdigit* *xs* *ys*) [0..*Suc* *n*])  
<proof>

**lemma** *num-sumdigit-eq-sum'*:  
**assumes** *symbols-lt* 4 *xs* **and** *symbols-lt* 4 *ys*  
**shows** *num* *xs* + *num* *ys* = *num* (*map* (*sumdigit* *xs* *ys*) [0..*Suc* (*max* (*length* *xs*) (*length* *ys*))])  
<proof>

**lemma** *num-sumdigit-eq-sum''*:  
**assumes** *bit-symbols* *xs* **and** *bit-symbols* *ys*  
**shows** *num* *xs* + *num* *ys* = *num* (*map* (*sumdigit* *xs* *ys*) [0..*Suc* (*max* (*length* *xs*) (*length* *ys*))])  
<proof>

**lemma** *sumdigit-bit-symbols*: *bit-symbols* (*map* (*sumdigit* *xs* *ys*) [0..*t*])  
<proof>

The core of the addition Turing machine is the following command. It scans the symbols on tape *j*<sub>1</sub> and *j*<sub>2</sub> in lockstep until it reaches blanks on both tapes. In every step it adds the symbols on both tapes and the symbol on the last tape, which is a memorization tape storing the carry bit. The sum of these three bits modulo 2 is written to tape *j*<sub>2</sub> and the new carry to the memorization tape.

**definition** *cmd-plus* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *command* **where**  
*cmd-plus* *j1* *j2* *rs*  $\equiv$   
(*if* *rs* ! *j1* = □  $\wedge$  *rs* ! *j2* = □ *then* 1 *else* 0,  
(*map* ( $\lambda$ *j*.  
*if* *j* = *j1* *then* (*rs* ! *j*, *Right*)  
*else if* *j* = *j2* *then* (*tosym* ((*todigit* (*rs* ! *j1*) + *todigit* (*rs* ! *j2*) + *todigit* (*last* *rs*)) mod 2), *Right*)  
*else if* *j* = *length* *rs* - 1 *then* (*tosym* ((*todigit* (*rs* ! *j1*) + *todigit* (*rs* ! *j2*) + *todigit* (*last* *rs*)) div 2), *Stay*)  
*else* (*rs* ! *j*, *Stay*)) [0..*length* *rs*]))

**lemma** *sem-cmd-plus*:  
**assumes** *j1*  $\neq$  *j2*  
**and** *j1* < *k* - 1

```

and  $j_2 < k - 1$ 
and  $j_2 > 0$ 
and  $\text{length } tps = k$ 
and  $\text{bit-symbols } xs$ 
and  $\text{bit-symbols } ys$ 
and  $tps ! j_1 = (\lfloor xs \rfloor, \text{Suc } t)$ 
and  $tps ! j_2 = (\lfloor \text{map } (\text{sumdigit } xs \text{ } ys) [0..<t] @ \text{drop } t \text{ } ys \rfloor, \text{Suc } t)$ 
and  $\text{last } tps = \lceil \text{carry } xs \text{ } ys \text{ } t \rceil$ 
and  $rs = \text{read } tps$ 
and  $tps' = tps$ 
   $[j_1 := tps ! j_1 \mid + \mid 1,$ 
   $j_2 := tps ! j_2 \mid := \mid \text{sumdigit } xs \text{ } ys \text{ } t \mid + \mid 1,$ 
   $\text{length } tps - 1 := \lceil \text{carry } xs \text{ } ys \text{ } (\text{Suc } t) \rceil]$ 
shows  $\text{sem } (\text{cmd-plus } j_1 \text{ } j_2) (0, tps) = (\text{if } t < \max (\text{length } xs) (\text{length } ys) \text{ then } 0 \text{ else } 1, tps')$ 
<proof>

```

**lemma** *contents-map-append-drop*:

```

 $\lfloor \text{map } f [0..<t] @ \text{drop } t \text{ } zs \rfloor (\text{Suc } t := f \text{ } t) = \lfloor \text{map } f [0..<\text{Suc } t] @ \text{drop } (\text{Suc } t) \text{ } zs \rfloor$ 
<proof>

```

**corollary** *sem-cmd-plus'*:

```

assumes  $j_1 \neq j_2$ 
  and  $j_1 < k - 1$ 
  and  $j_2 < k - 1$ 
  and  $j_2 > 0$ 
  and  $\text{length } tps = k$ 
  and  $\text{bit-symbols } xs$ 
  and  $\text{bit-symbols } ys$ 
  and  $tps ! j_1 = (\lfloor xs \rfloor, \text{Suc } t)$ 
  and  $tps ! j_2 = (\lfloor \text{map } (\text{sumdigit } xs \text{ } ys) [0..<t] @ \text{drop } t \text{ } ys \rfloor, \text{Suc } t)$ 
  and  $\text{last } tps = \lceil \text{carry } xs \text{ } ys \text{ } t \rceil$ 
  and  $tps' = tps$ 
   $[j_1 := (\lfloor xs \rfloor, \text{Suc } (\text{Suc } t)),$ 
   $j_2 := (\lfloor \text{map } (\text{sumdigit } xs \text{ } ys) [0..<\text{Suc } t] @ \text{drop } (\text{Suc } t) \text{ } ys \rfloor, \text{Suc } (\text{Suc } t)),$ 
   $\text{length } tps - 1 := \lceil \text{carry } xs \text{ } ys \text{ } (\text{Suc } t) \rceil]$ 
shows  $\text{sem } (\text{cmd-plus } j_1 \text{ } j_2) (0, tps) = (\text{if } \text{Suc } t \leq \max (\text{length } xs) (\text{length } ys) \text{ then } 0 \text{ else } 1, tps')$ 
<proof>

```

The next Turing machine comprises just the command *cmd-plus*. It overwrites tape  $j_2$  with the sum of the numbers on tape  $j_1$  and  $j_2$ . The carry bit is maintained on the last tape.

**definition** *tm-plus* ::  $\text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**

```

 $\text{tm-plus } j_1 \text{ } j_2 \equiv [\text{cmd-plus } j_1 \text{ } j_2]$ 

```

**lemma** *tm-plus-tm*:

```

assumes  $j_2 > 0$  and  $k \geq 2$  and  $G \geq 4$ 
shows  $\text{turing-machine } k \text{ } G (\text{tm-plus } j_1 \text{ } j_2)$ 
<proof>

```

**lemma** *tm-plus-immobile*:

```

fixes  $k :: \text{nat}$ 
assumes  $j_1 < k$  and  $j_2 < k$ 
shows  $\text{immobile } (\text{tm-plus } j_1 \text{ } j_2) k (\text{Suc } k)$ 
<proof>

```

**lemma** *execute-tm-plus*:

```

assumes  $j_1 \neq j_2$ 
  and  $j_1 < k - 1$ 
  and  $j_2 < k - 1$ 
  and  $j_2 > 0$ 
  and  $\text{length } tps = k$ 
  and  $\text{bit-symbols } xs$ 
  and  $\text{bit-symbols } ys$ 
  and  $t \leq \text{Suc } (\max (\text{length } xs) (\text{length } ys))$ 

```

**and**  $tps ! j1 = (\lfloor xs \rfloor, 1)$   
**and**  $tps ! j2 = (\lfloor ys \rfloor, 1)$   
**and**  $last\ tps = \lceil \triangleright \rceil$   
**shows**  $execute\ (tm-plus\ j1\ j2)\ (0, tps)\ t =$   
*(if  $t \leq \max\ (length\ xs)\ (length\ ys)$  then 0 else 1, tps*  
 $\lfloor j1 := (\lfloor xs \rfloor, Suc\ t),$   
 $j2 := (\lfloor map\ (sumdigit\ xs\ ys)\ [0..<t] \rceil @\ drop\ t\ ys \rfloor, Suc\ t),$   
 $length\ tps - 1 := \lceil carry\ xs\ ys\ t \rceil$ )  
*(proof)*

**lemma** *tm-plus-bounded-write:*  
**assumes**  $j1 < k - 1$   
**shows**  $bounded-write\ (tm-plus\ j1\ j2)\ (k - 1)\ 4$   
*(proof)*

**lemma** *carry-max-length:*  
**assumes** *bit-symbols xs and bit-symbols ys*  
**shows**  $carry\ xs\ ys\ (Suc\ (\max\ (length\ xs)\ (length\ ys))) = 0$   
*(proof)*

**corollary** *execute-tm-plus-halt:*  
**assumes**  $j1 \neq j2$   
**and**  $j1 < k - 1$   
**and**  $j2 < k - 1$   
**and**  $j2 > 0$   
**and**  $length\ tps = k$   
**and** *bit-symbols xs*  
**and** *bit-symbols ys*  
**and**  $t = Suc\ (\max\ (length\ xs)\ (length\ ys))$   
**and**  $tps ! j1 = (\lfloor xs \rfloor, 1)$   
**and**  $tps ! j2 = (\lfloor ys \rfloor, 1)$   
**and**  $last\ tps = \lceil \triangleright \rceil$   
**shows**  $execute\ (tm-plus\ j1\ j2)\ (0, tps)\ t =$   
 $(1, tps$   
 $\lfloor j1 := (\lfloor xs \rfloor, Suc\ t),$   
 $j2 := (\lfloor map\ (sumdigit\ xs\ ys)\ [0..<t] \rceil, Suc\ t),$   
 $length\ tps - 1 := \lceil 0 \rceil$ )  
*(proof)*

**lemma** *transforms-tm-plusI:*  
**assumes**  $j1 \neq j2$   
**and**  $j1 < k - 1$   
**and**  $j2 < k - 1$   
**and**  $j2 > 0$   
**and**  $length\ tps = k$   
**and** *bit-symbols xs*  
**and** *bit-symbols ys*  
**and**  $t = Suc\ (\max\ (length\ xs)\ (length\ ys))$   
**and**  $tps ! j1 = (\lfloor xs \rfloor, 1)$   
**and**  $tps ! j2 = (\lfloor ys \rfloor, 1)$   
**and**  $last\ tps = \lceil \triangleright \rceil$   
**and**  $tps' = tps$   
 $\lfloor j1 := (\lfloor xs \rfloor, Suc\ t),$   
 $j2 := (\lfloor map\ (sumdigit\ xs\ ys)\ [0..<t] \rceil, Suc\ t),$   
 $length\ tps - 1 := \lceil 0 \rceil$   
**shows**  $transforms\ (tm-plus\ j1\ j2)\ tps\ t\ tps'$   
*(proof)*

The next Turing machine removes the memorization tape from *tm-plus*.

**definition**  $tm-plus' :: tapeidx \Rightarrow tapeidx \Rightarrow machine$  **where**  
 $tm-plus'\ j1\ j2 \equiv cartesian\ (tm-plus\ j1\ j2)\ 4$

**lemma** *tm-plus'-tm:*

assumes  $j2 > 0$  and  $k \geq 2$  and  $G \geq 4$   
 shows *turing-machine*  $k$   $G$  (*tm-plus'*  $j1$   $j2$ )  
 ⟨*proof*⟩

**lemma** *transforms-tm-plus'I* [*transforms-intros*]:

fixes  $k$   $t :: \text{nat}$  and  $j1$   $j2 :: \text{tapeidx}$  and  $tps$   $tps' :: \text{tape list}$  and  $xs$   $zs :: \text{symbol list}$   
 assumes  $j1 \neq j2$   
 and  $j1 < k$   
 and  $j2 < k$   
 and  $j2 > 0$   
 and  $\text{length } tps = k$   
 and *bit-symbols*  $xs$   
 and *bit-symbols*  $ys$   
 and  $t = \text{Suc } (\text{max } (\text{length } xs) (\text{length } ys))$   
 and  $tps ! j1 = (\lfloor xs \rfloor, 1)$   
 and  $tps ! j2 = (\lfloor ys \rfloor, 1)$   
 and  $tps' = tps$   
 $j1 := (\lfloor xs \rfloor, \text{Suc } t)$ ,  
 $j2 := (\lfloor \text{map } (\text{sumdigit } xs \ ys) [0..<t] \rfloor, \text{Suc } t)$   
 shows *transforms* (*tm-plus'*  $j1$   $j2$ )  $tps$   $t$   $tps'$   
 ⟨*proof*⟩

The next Turing machine is the one we actually use to add two numbers. After computing the sum by running *tm-plus'*, it removes trailing zeros and performs a carriage return on the tapes  $j_1$  and  $j_2$ .

**definition** *tm-add* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-add*  $j1$   $j2 \equiv$   
*tm-plus'*  $j1$   $j2$  ;;  
*tm-lconst-until*  $j2$   $j2$   $\{h. h \neq \mathbf{0} \wedge h \neq \square\}$   $\square$  ;;  
*tm-cr*  $j1$  ;;  
*tm-cr*  $j2$

**lemma** *tm-add-tm*:

assumes  $j2 > 0$  and  $k \geq 2$  and  $G \geq 4$  and  $j2 < k$   
 shows *turing-machine*  $k$   $G$  (*tm-add*  $j1$   $j2$ )  
 ⟨*proof*⟩

**locale** *turing-machine-add* =

fixes  $j1$   $j2 :: \text{tapeidx}$

**begin**

**definition**  $tm1 \equiv \text{tm-plus}' j1 j2$

**definition**  $tm2 \equiv tm1$  ;; *tm-lconst-until*  $j2$   $j2$   $\{h. h \neq \mathbf{0} \wedge h \neq \square\}$   $\square$

**definition**  $tm3 \equiv tm2$  ;; *tm-cr*  $j1$

**definition**  $tm4 \equiv tm3$  ;; *tm-cr*  $j2$

**lemma** *tm4-eq-tm-add*:  $tm4 = \text{tm-add } j1 j2$

⟨*proof*⟩

**context**

fixes  $x$   $y$   $k :: \text{nat}$  and  $tps0 :: \text{tape list}$

assumes  $jk$ :  $j1 \neq j2$   $j1 < k$   $j2 < k$   $j2 > 0$   $k = \text{length } tps0$

assumes  $tps0$ :

$tps0 ! j1 = (\lfloor \text{canrepr } x \rfloor, 1)$

$tps0 ! j2 = (\lfloor \text{canrepr } y \rfloor, 1)$

**begin**

**abbreviation**  $xs \equiv \text{canrepr } x$

**abbreviation**  $ys \equiv \text{canrepr } y$

**lemma**  $xs$ : *bit-symbols*  $xs$

⟨*proof*⟩

**lemma** *ys: bit-symbols ys*  
⟨*proof*⟩

**abbreviation**  $n \equiv \text{Suc } (\max (\text{length } xs) (\text{length } ys))$

**abbreviation**  $m \equiv \text{length } (\text{canrepr } (\text{num } xs + \text{num } ys))$

**definition**  $tps1 \equiv tps0$

$[j1 := (\lfloor xs \rfloor, \text{Suc } n),$   
 $j2 := (\lfloor \text{map } (\text{sumdigit } xs \text{ } ys) [0..<n] \rfloor, \text{Suc } n)]$

**lemma** *tm1 [transforms-intros]:*  
**assumes**  $ttt = n$   
**shows** *transforms tm1 tps0 ttt tps1*  
⟨*proof*⟩

**definition**  $tps2 \equiv tps0$

$[j1 := (\lfloor xs \rfloor, \text{Suc } n),$   
 $j2 := (\lfloor \text{canrepr } (\text{num } xs + \text{num } ys) \rfloor, m)]$

**lemma** *contents-canlen:*  
**assumes** *bit-symbols zs*  
**shows**  $\lfloor zs \rfloor (\text{canlen } zs) \in \{h. h \neq \mathbf{0} \wedge \square < h\}$   
⟨*proof*⟩

**lemma** *tm2 [transforms-intros]:*  
**assumes**  $ttt = n + \text{Suc } (\text{Suc } n - \text{canlen } (\text{map } (\text{sumdigit } xs \text{ } ys) [0..<n]))$   
**shows** *transforms tm2 tps0 ttt tps2*  
⟨*proof*⟩

**definition**  $tps3 \equiv tps0$

$[j1 := (\lfloor xs \rfloor, 1),$   
 $j2 := (\lfloor \text{canrepr } (\text{num } xs + \text{num } ys) \rfloor, m)]$

**lemma** *tm3 [transforms-intros]:*  
**assumes**  $ttt = n + \text{Suc } (\text{Suc } n - \text{canlen } (\text{map } (\text{sumdigit } xs \text{ } ys) [0..<n])) + \text{Suc } n + 2$   
**shows** *transforms tm3 tps0 ttt tps3*  
⟨*proof*⟩

**definition**  $tps4 \equiv tps0$

$[j1 := (\lfloor xs \rfloor, 1),$   
 $j2 := (\lfloor \text{canrepr } (\text{num } xs + \text{num } ys) \rfloor, 1)]$

**lemma** *tm4:*  
**assumes**  $ttt = n + \text{Suc } (\text{Suc } n - \text{canlen } (\text{map } (\text{sumdigit } xs \text{ } ys) [0..<n])) + \text{Suc } n + 2 + m + 2$   
**shows** *transforms tm4 tps0 ttt tps4*  
⟨*proof*⟩

**lemma** *tm4':*  
**assumes**  $ttt = 3 * \max (\text{length } xs) (\text{length } ys) + 10$   
**shows** *transforms tm4 tps0 ttt tps4*  
⟨*proof*⟩

**definition**  $tps4' \equiv tps0$   
 $[j2 := (\lfloor x + y \rfloor_N, 1)]$

**lemma** *tm4'':*  
**assumes**  $ttt = 3 * \max (\text{nlength } x) (\text{nlength } y) + 10$   
**shows** *transforms tm4 tps0 ttt tps4'*  
⟨*proof*⟩

**end**

end

**lemma** *transforms-tm-addI* [*transforms-intros*]:  
**fixes**  $j1\ j2 :: \text{tapeid}x$   
**fixes**  $x\ y\ k\ ttt :: \text{nat}$  **and**  $tps\ tps' :: \text{tape list}$   
**assumes**  $j1 \neq j2\ j1 < k\ j2 < k\ j2 > 0\ k = \text{length } tps$   
**assumes**  
 $tps ! j1 = (\lfloor \text{canrepr } x \rfloor, 1)$   
 $tps ! j2 = (\lfloor \text{canrepr } y \rfloor, 1)$   
**assumes**  $ttt = 3 * \max (\text{nlength } x) (\text{nlength } y) + 10$   
**assumes**  $tps' = tps$   
 $[j2 := (\lfloor x + y \rfloor_N, 1)]$   
**shows** *transforms (tm-add j1 j2) tps ttt tps'*  
*<proof>*

## 2.7.5 Multiplication

In this section we construct a Turing machine that multiplies two numbers, each on its own tape, and writes the result to another tape. It employs the common algorithm for multiplication, which for binary numbers requires only doubling a number and adding two numbers. For the latter we already have a TM; for the former we are going to construct one.

### The common algorithm

For two numbers given as symbol sequences  $xs$  and  $ys$ , the common algorithm maintains an intermediate result, initialized with 0, and scans  $xs$  starting from the most significant digit. In each step the intermediate result is multiplied by two, and if the current digit of  $xs$  is **1**, the value of  $ys$  is added to the intermediate result.

**fun** *prod* :: *symbol list*  $\Rightarrow$  *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**  
 $\text{prod } xs\ ys\ 0 = 0 \mid$   
 $\text{prod } xs\ ys\ (\text{Suc } i) = 2 * \text{prod } xs\ ys\ i + (\text{if } xs ! (\text{length } xs - 1 - i) = 3 \text{ then } \text{num } ys \text{ else } 0)$

After  $i$  steps of the algorithm, the intermediate result is the product of  $ys$  and the  $i$  most significant bits of  $xs$ .

**lemma** *prod*:  
**assumes**  $i \leq \text{length } xs$   
**shows**  $\text{prod } xs\ ys\ i = \text{num } (\text{drop } (\text{length } xs - i) xs) * \text{num } ys$   
*<proof>*

After  $\text{length } xs$  steps, the intermediate result is the final result:

**corollary** *prod-eq-prod*:  $\text{prod } xs\ ys\ (\text{length } xs) = \text{num } xs * \text{num } ys$   
*<proof>*

**definition** *prod'* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**  
 $\text{prod}'\ x\ y\ i \equiv \text{prod } (\text{canrepr } x) (\text{canrepr } y)\ i$

**lemma** *prod'*:  $\text{prod}'\ x\ y\ (\text{nlength } x) = x * y$   
*<proof>*

### Multiplying by two

Since we represent numbers with the least significant bit at the left, a multiplication by two is a right shift with a **0** inserted as the least significant digit. The next command implements the right shift. It scans the tape  $j$  and memorizes the current symbol on the last tape. It only writes the symbols **0** and **1**.

**definition** *cmd-double* :: *tapeid*  $\Rightarrow$  *command* **where**  
 $\text{cmd-double } j\ rs \equiv$   
 $(\text{if } rs ! j = \square \text{ then } 1 \text{ else } 0,$   
 $\text{map } (\lambda i.$   
 $\text{if } i = j \text{ then}$   
 $\text{if last } rs = \triangleright \wedge rs ! j = \square \text{ then } (rs ! i, \text{Right})$

else (tosym (todigit (last rs)), Right)  
 else if i = length rs - 1 then (tosym (todigit (rs ! j)), Stay)  
 else (rs ! i, Stay) [0..<length rs])

**lemma** *turing-command-double*:

assumes  $k \geq 2$  and  $G \geq 4$  and  $j > 0$  and  $j < k - 1$   
 shows *turing-command*  $k$  1  $G$  (*cmd-double*  $j$ )

*<proof>*

**lemma** *sem-cmd-double-0*:

assumes  $j < k$   
 and *bit-symbols*  $xs$   
 and  $i \leq \text{length } xs$   
 and  $i > 0$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $tps ! j = (\lfloor xs \rfloor, i)$   
 and  $tps ! k = \lceil z \rceil$   
 and  $tps' = tps [j := tps ! j |:=| \text{tosym } (\text{todigit } z) \mid+ \mid 1, k := \lceil xs ! (i - 1) \rceil]$   
 shows *sem* (*cmd-double*  $j$ )  $(0, tps) = (0, tps')$

*<proof>*

**lemma** *sem-cmd-double-1*:

assumes  $j < k$   
 and *bit-symbols*  $xs$   
 and  $i > \text{length } xs$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $tps ! j = (\lfloor xs \rfloor, i)$   
 and  $tps ! k = \lceil z \rceil$   
 and  $tps' = tps$   
 $[j := tps ! j |:=| (\text{if } z = \triangleright \text{ then } \square \text{ else } \text{tosym } (\text{todigit } z)) \mid+ \mid 1,$   
 $k := \lceil 0 \rceil]$   
 shows *sem* (*cmd-double*  $j$ )  $(0, tps) = (1, tps')$

*<proof>*

The next Turing machine consists just of the command *cmd-double*.

**definition** *tm-double* :: *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-double*  $j \equiv [\text{cmd-double } j]$

**lemma** *tm-double-tm*:

assumes  $k \geq 2$  and  $G \geq 4$  and  $j > 0$  and  $j < k - 1$   
 shows *turing-machine*  $k$   $G$  (*tm-double*  $j$ )

*<proof>*

**lemma** *execute-tm-double-0*:

assumes  $j < k$   
 and *bit-symbols*  $xs$   
 and  $\text{length } xs > 0$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $tps ! j = (\lfloor xs \rfloor, 1)$   
 and  $tps ! k = \lceil \triangleright \rceil$   
 and  $t \geq 1$   
 and  $t \leq \text{length } xs$   
 shows *execute* (*tm-double*  $j$ )  $(0, tps) t =$   
 $(0, tps [j := (\lfloor 0 \# \text{take } (t - 1) \text{ } xs \text{ @ drop } t \text{ } xs \rfloor, \text{Suc } t), k := \lceil xs ! (t - 1) \rceil])$

*<proof>*

**lemma** *execute-tm-double-1*:

assumes  $j < k$   
 and *bit-symbols*  $xs$   
 and  $\text{length } xs > 0$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $tps ! j = (\lfloor xs \rfloor, 1)$   
 and  $tps ! k = \lceil \triangleright \rceil$

**shows** *execute* (*tm-double j*) (*0*, *tps*) (*Suc (length xs)*) =  
 (*1*, *tps [j := ([0 # xs], length xs + 2), k := [0]]*)  
 ⟨*proof*⟩

**lemma** *execute-tm-double-Nil*:

**assumes**  $j < k$   
**and**  $\text{length } tps = \text{Suc } k$   
**and**  $tps ! j = ([\ ], 1)$   
**and**  $tps ! k = [\triangleright]$   
**shows** *execute* (*tm-double j*) (*0*, *tps*) (*Suc 0*) =  
 (*1*, *tps [j := ([\ ], 2), k := [0]]*)  
 ⟨*proof*⟩

**lemma** *execute-tm-double*:

**assumes**  $j < k$   
**and**  $\text{length } tps = \text{Suc } k$   
**and**  $tps ! j = (\lfloor \text{canrepr } n \rfloor, 1)$   
**and**  $tps ! k = [\triangleright]$   
**shows** *execute* (*tm-double j*) (*0*, *tps*) (*Suc (length (canrepr n))*) =  
 (*1*, *tps [j := (\lfloor canrepr (2 \* n) \rfloor, length (canrepr n) + 2), k := [0]]*)  
 ⟨*proof*⟩

**lemma** *execute-tm-double-app*:

**assumes**  $j < k$   
**and**  $\text{length } tps = k$   
**and**  $tps ! j = (\lfloor \text{canrepr } n \rfloor, 1)$   
**shows** *execute* (*tm-double j*) (*0*, *tps @ [[\triangleright]]*) (*Suc (length (canrepr n))*) =  
 (*1*, *tps [j := (\lfloor canrepr (2 \* n) \rfloor, length (canrepr n) + 2)] @ [[0]]*)  
 ⟨*proof*⟩

**lemma** *transforms-tm-double*:

**assumes**  $j < k$   
**and**  $\text{length } tps = k$   
**and**  $tps ! j = (\lfloor \text{canrepr } n \rfloor, 1)$   
**shows** *transforms* (*tm-double j*)  
 (*tps @ [[\triangleright]]*)  
 (*Suc (length (canrepr n))*)  
 (*tps [j := (\lfloor canrepr (2 \* n) \rfloor, length (canrepr n) + 2)] @ [[0]]*)  
 ⟨*proof*⟩

**lemma** *tm-double-immobile*:

**fixes**  $k :: \text{nat}$   
**assumes**  $j > 0$  **and**  $j < k$   
**shows** *immobile* (*tm-double j*)  $k$  (*Suc k*)  
 ⟨*proof*⟩

**lemma** *tm-double-bounded-write*:

**assumes**  $j < k - 1$   
**shows** *bounded-write* (*tm-double j*) ( $k - 1$ )  $4$   
 ⟨*proof*⟩

The next Turing machine removes the memorization tape.

**definition** *tm-double'* ::  $\text{nat} \Rightarrow \text{machine}$  **where**

*tm-double' j*  $\equiv$  *cartesian* (*tm-double j*)  $4$

**lemma** *tm-double'-tm*:

**assumes**  $j > 0$  **and**  $k \geq 2$  **and**  $G \geq 4$  **and**  $j < k$   
**shows** *turing-machine*  $k$   $G$  (*tm-double' j*)  
 ⟨*proof*⟩

**lemma** *transforms-tm-double'I* [*transforms-intros*]:

**assumes**  $j > 0$  **and**  $j < k$   
**and**  $\text{length } tps = k$

```

and tps ! j = ([canrepr n], 1)
and t = (Suc (length (canrepr n)))
and tps' = tps [j := ([canrepr (2 * n)], length (canrepr n) + 2)]
shows transforms (tm-double' j) tps t tps'
⟨proof⟩

```

The next Turing machine is the one we actually use to double a number. It runs *tm-double'* and performs a carriage return.

**definition** *tm-times2* :: *tapeidx*  $\Rightarrow$  *machine* **where**  
*tm-times2* j  $\equiv$  *tm-double'* j ;; *tm-cr* j

**lemma** *tm-times2-tm*:  
**assumes**  $k \geq 2$  **and**  $j > 0$  **and**  $j < k$  **and**  $G \geq 4$   
**shows** *turing-machine* k G (*tm-times2* j)  
⟨proof⟩

**lemma** *transforms-tm-times2I* [*transforms-intros*]:  
**assumes**  $j > 0$  **and**  $j < k$   
**and** *length* tps = k  
**and** tps ! j = ([n]<sub>N</sub>, 1)  
**and** t = 5 + 2 \* *nlength* n  
**and** tps' = tps [j := ([2 \* n]<sub>N</sub>, 1)]  
**shows** *transforms* (*tm-times2* j) tps t tps'  
⟨proof⟩

## Multiplying arbitrary numbers

Before we can multiply arbitrary numbers we need just a few more lemmas.

**lemma** *num-drop-le-nu*: *num* (*drop* j *xs*)  $\leq$  *num* *xs*  
⟨proof⟩

**lemma** *nlength-prod-le-prod*:  
**assumes**  $i \leq$  *length* *xs*  
**shows** *nlength* (*prod* *xs* *ys* i)  $\leq$  *nlength* (*num* *xs* \* *num* *ys*)  
⟨proof⟩

**corollary** *nlength-prod'-le-prod*:  
**assumes**  $i \leq$  *nlength* *x*  
**shows** *nlength* (*prod'* *x* *y* i)  $\leq$  *nlength* (*x* \* *y*)  
⟨proof⟩

**lemma** *two-times-prod*:  
**assumes**  $i <$  *length* *xs*  
**shows** 2 \* *prod* *xs* *ys* i  $\leq$  *num* *xs* \* *num* *ys*  
⟨proof⟩

**corollary** *two-times-prod'*:  
**assumes**  $i <$  *nlength* *x*  
**shows** 2 \* *prod'* *x* *y* i  $\leq$  *x* \* *y*  
⟨proof⟩

The next Turing machine multiplies the numbers on tapes  $j_1$  and  $j_2$  and writes the result to tape  $j_3$ . It iterates over the binary digits on  $j_1$  starting from the most significant digit. In each iteration it doubles the intermediate result on  $j_3$ . If the current digit is **1**, the number on  $j_2$  is added to  $j_3$ .

**definition** *tm-mult* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**  
*tm-mult* j1 j2 j3  $\equiv$   
*tm-right-until* j1 {□} ;;  
*tm-left* j1 ;;  
*WHILE* [] ;  $\lambda$ rs. rs ! j1  $\neq$  ▷ *DO*  
*tm-times2* j3 ;;  
*IF*  $\lambda$ rs. rs ! j1 = **1** *THEN*

```

    tm-add j2 j3
  ELSE
  []
  ENDIF ;;
  tm-left j1
  DONE ;;
  tm-right j1

```

**lemma** *tm-mult-tm*:

```

  assumes j1 ≠ j2 j2 ≠ j3 j3 ≠ j1 and j3 > 0
  assumes k ≥ 2
    and G ≥ 4
    and j1 < k j2 < k j3 < k
  shows turing-machine k G (tm-mult j1 j2 j3)
  ⟨proof⟩

```

**locale** *turing-machine-mult* =

```

  fixes j1 j2 j3 :: tapeidx
  begin

```

**definition** *tm1* ≡ *tm-right-until* j1 {□}

**definition** *tm2* ≡ *tm1* ;; *tm-left* j1

**definition** *tmIf* ≡ IF λrs. rs ! j1 = 1 THEN *tm-add* j2 j3 ELSE [] ENDIF

**definition** *tmBody1* ≡ *tm-times2* j3 ;; *tmIf*

**definition** *tmBody* ≡ *tmBody1* ;; *tm-left* j1

**definition** *tmWhile* ≡ WHILE [] ; λrs. rs ! j1 ≠ ▷ DO *tmBody* DONE

**definition** *tm3* ≡ *tm2* ;; *tmWhile*

**definition** *tm4* ≡ *tm3* ;; *tm-right* j1

**lemma** *tm4-eq-tm-mult*: *tm4* = *tm-mult* j1 j2 j3

⟨proof⟩

**context**

fixes x y k :: nat and tps0 :: tape list

assumes jk: j1 ≠ j2 j2 ≠ j3 j3 ≠ j1 j3 > 0 j1 < k j2 < k j3 < k length tps0 = k

assumes tps0:

tps0 ! j1 = ([x]<sub>N</sub>, 1)

tps0 ! j2 = ([y]<sub>N</sub>, 1)

tps0 ! j3 = ([0]<sub>N</sub>, 1)

**begin**

**definition** *tps1* ≡ *tps0* [j1 := ([x]<sub>N</sub>, Suc (nlength x))]

**lemma** *tm1* [*transforms-intros*]:

assumes t = Suc (nlength x)

shows *transforms* *tm1* *tps0* t *tps1*

⟨proof⟩

**definition** *tps2* ≡ *tps0* [j1 := ([x]<sub>N</sub>, nlength x)]

**lemma** *tm2* [*transforms-intros*]:

assumes t = Suc (Suc (nlength x)) and tps' = *tps2*

shows *transforms* *tm2* *tps0* t *tps'*

⟨proof⟩

**definition** *tpsL* t ≡ *tps0*

[j1 := ([x]<sub>N</sub>, nlength x - t),

j3 := ([prod' x y t]<sub>N</sub>, 1)]

**definition** *tpsL1* t ≡ *tps0*

[j1 := ([x]<sub>N</sub>, nlength x - t),

j3 := ([2 \* prod' x y t]<sub>N</sub>, 1)]

**definition**  $tpsL2\ t \equiv tps0$   
 $[j1 := (\lfloor x \rfloor_N, nlength\ x - t),$   
 $j3 := (\lfloor prod' x\ y\ (Suc\ t) \rfloor_N, 1)]$

**definition**  $tpsL3\ t \equiv tps0$   
 $[j1 := (\lfloor x \rfloor_N, nlength\ x - t - 1),$   
 $j3 := (\lfloor prod' x\ y\ (Suc\ t) \rfloor_N, 1)]$

**lemma**  $tmIf$  [*transforms-intros*]:  
**assumes**  $t < nlength\ x$  **and**  $ttt = 12 + 3 * nlength\ (x * y)$   
**shows**  $transforms\ tmIf\ (tpsL1\ t)\ ttt\ (tpsL2\ t)$   
*<proof>*

**lemma**  $tmBody1$  [*transforms-intros*]:  
**assumes**  $t < nlength\ x$   
**and**  $ttt = 17 + 2 * nlength\ (Arithmetic.prod' x\ y\ t) + 3 * nlength\ (x * y)$   
**shows**  $transforms\ tmBody1\ (tpsL\ t)\ ttt\ (tpsL2\ t)$   
*<proof>*

**lemma**  $tmBody$ :  
**assumes**  $t < nlength\ x$   
**and**  $ttt = 6 + 2 * nlength\ (prod' x\ y\ t) + (12 + 3 * nlength\ (x * y))$   
**shows**  $transforms\ tmBody\ (tpsL\ t)\ ttt\ (tpsL\ (Suc\ t))$   
*<proof>*

**lemma**  $tmBody'$  [*transforms-intros*]:  
**assumes**  $t < nlength\ x$  **and**  $ttt = 18 + 5 * nlength\ (x * y)$   
**shows**  $transforms\ tmBody'\ (tpsL\ t)\ ttt\ (tpsL\ (Suc\ t))$   
*<proof>*

**lemma**  $read$ -contents:  
**fixes**  $tps :: tape\ list$  **and**  $j :: tapeidx$  **and**  $zs :: symbol\ list$   
**assumes**  $tps\ !\ j = (\lfloor zs \rfloor, i)$  **and**  $i > 0$  **and**  $i \leq length\ zs$  **and**  $j < length\ tps$   
**shows**  $read\ tps\ !\ j = zs\ !\ (i - 1)$   
*<proof>*

**lemma**  $tmWhile$  [*transforms-intros*]:  
**assumes**  $ttt = 1 + 25 * (nlength\ x + nlength\ y) * (nlength\ x + nlength\ y)$   
**shows**  $transforms\ tmWhile\ (tpsL\ 0)\ ttt\ (tpsL\ (nlength\ x))$   
*<proof>*

**lemma**  $tm3$ :  
**assumes**  $ttt = Suc\ (Suc\ (nlength\ x)) +$   
 $Suc\ ((25 * nlength\ x + 25 * nlength\ y) * (nlength\ x + nlength\ y))$   
**shows**  $transforms\ tm3\ tps0\ ttt\ (tpsL\ (nlength\ x))$   
*<proof>*

**definition**  $tps3 \equiv tps0$   
 $[j1 := (\lfloor x \rfloor_N, 0),$   
 $j3 := (\lfloor x * y \rfloor_N, 1)]$

**lemma**  $tm3'$  [*transforms-intros*]:  
**assumes**  $ttt = 3 + 26 * (nlength\ x + nlength\ y) * (nlength\ x + nlength\ y)$   
**shows**  $transforms\ tm3'\ tps0\ ttt\ tps3$   
*<proof>*

**definition**  $tps4 \equiv tps0$   
 $[j3 := (\lfloor x * y \rfloor_N, 1)]$

**lemma**  $tm4$ :  
**assumes**  $ttt = 4 + 26 * (nlength\ x + nlength\ y) * (nlength\ x + nlength\ y)$   
**shows**  $transforms\ tm4\ tps0\ ttt\ tps4$   
*<proof>*

end

end

**lemma** *transforms-tm-mult* [*transforms-intros*]:  
fixes  $j1\ j2\ j3 :: \text{tapeidx}$  and  $x\ y\ k\ ttt :: \text{nat}$  and  $tps\ tps' :: \text{tape list}$   
assumes  $j1 \neq j2\ j2 \neq j3\ j3 \neq j1\ j3 > 0$   
assumes  $\text{length } tps = k$   
and  $j1 < k\ j2 < k\ j3 < k$   
and  $tps ! j1 = (\lfloor x \rfloor_N, 1)$   
and  $tps ! j2 = (\lfloor y \rfloor_N, 1)$   
and  $tps ! j3 = (\lfloor 0 \rfloor_N, 1)$   
and  $ttt = 4 + 26 * (\text{nlength } x + \text{nlength } y) * (\text{nlength } x + \text{nlength } y)$   
and  $tps' = tps [j3 := (\lfloor x * y \rfloor_N, 1)]$   
shows *transforms* (*tm-mult*  $j1\ j2\ j3$ )  $tps\ ttt\ tps'$   
(*proof*)

## 2.7.6 Powers

In this section we construct for every  $d \in \mathbb{N}$  a Turing machine that computes  $n^d$ . The following TMs expect a number  $n$  on tape  $j_1$  and output  $n^d$  on tape  $j_3$ . Another tape,  $j_2$ , is used as scratch space to hold intermediate values. The TMs initialize tape  $j_3$  with 1 and then multiply this value by  $n$  for  $d$  times using the TM *tm-mult*.

**fun** *tm-pow* ::  $\text{nat} \Rightarrow \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**  
*tm-pow* 0  $j1\ j2\ j3 = \text{tm-setn } j3\ 1 \mid$   
*tm-pow* (*Suc*  $d$ )  $j1\ j2\ j3 =$   
*tm-pow*  $d\ j1\ j2\ j3 ;; (\text{tm-copyn } j3\ j2 ;; \text{tm-setn } j3\ 0 ;; \text{tm-mult } j1\ j2\ j3 ;; \text{tm-setn } j2\ 0)$

**lemma** *tm-pow-tm*:  
assumes  $j1 \neq j2\ j2 \neq j3\ j3 \neq j1$   
and  $0 < j2\ 0 < j3\ 0 < j1$   
assumes  $j1 < k\ j2 < k\ j3 < k$   
and  $k \geq 2$   
and  $G \geq 4$   
shows *turing-machine*  $k\ G$  (*tm-pow*  $d\ j1\ j2\ j3$ )  
(*proof*)

**locale** *turing-machine-pow* =  
fixes  $j1\ j2\ j3 :: \text{tapeidx}$   
**begin**

**definition**  $tm1 \equiv \text{tm-copyn } j3\ j2 ;; \text{tm-setn } j3\ 0$   
**definition**  $tm2 \equiv tm1 ;; \text{tm-mult } j1\ j2\ j3$   
**definition**  $tm3 \equiv tm2 ;; \text{tm-setn } j2\ 0$

**fun**  $tm4 :: \text{nat} \Rightarrow \text{machine}$  **where**  
 $tm4\ 0 = \text{tm-setn } j3\ 1 \mid$   
 $tm4$  (*Suc*  $d$ ) =  $tm4\ d ;; tm3$

**lemma** *tm4-eq-tm-pow*:  $tm4\ d = \text{tm-pow } d\ j1\ j2\ j3$   
(*proof*)

**context**  
fixes  $x\ y\ k :: \text{nat}$  and  $tps0 :: \text{tape list}$   
assumes  $jk: k = \text{length } tps0\ j1 < k\ j2 < k\ j3 < k$   
 $j1 \neq j2\ j2 \neq j3\ j3 \neq j1$   
 $0 < j2\ 0 < j3\ 0 < j1$   
assumes *tps0*:  
 $tps0 ! j1 = (\lfloor x \rfloor_N, 1)$   
 $tps0 ! j2 = (\lfloor 0 \rfloor_N, 1)$   
 $tps0 ! j3 = (\lfloor y \rfloor_N, 1)$   
**begin**

**definition**  $tps1 \equiv tps0$   
 $[j2 := (\lfloor y \rfloor_N, 1), j3 := (\lfloor 0 \rfloor_N, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:  
**assumes**  $ttt = 24 + 5 * nlength\ y$   
**shows** *transforms*  $tm1\ tps0\ ttt\ tps1$   
 $\langle proof \rangle$

**definition**  $tps2 \equiv tps0$   
 $[j2 := (\lfloor y \rfloor_N, 1),$   
 $j3 := (\lfloor x * y \rfloor_N, 1)]$

**lemma**  $tm2$  [*transforms-intros*]:  
**assumes**  $ttt = 28 + 5 * nlength\ y + (26 * nlength\ x + 26 * nlength\ y) * (nlength\ x + nlength\ y)$   
**shows** *transforms*  $tm2\ tps0\ ttt\ tps2$   
 $\langle proof \rangle$

**definition**  $tps3 \equiv tps0$   
 $[j3 := (\lfloor x * y \rfloor_N, 1)]$

**lemma**  $tm3$ :  
**assumes**  $ttt = 38 + 7 * nlength\ y + (26 * nlength\ x + 26 * nlength\ y) * (nlength\ x + nlength\ y)$   
**shows** *transforms*  $tm3\ tps0\ ttt\ tps3$   
 $\langle proof \rangle$

**lemma**  $tm3'$ :  
**assumes**  $ttt = 38 + 33 * (nlength\ x + nlength\ y) ^ 2$   
**shows** *transforms*  $tm3\ tps0\ ttt\ tps3$   
 $\langle proof \rangle$

**end**

**lemma**  $tm3''$  [*transforms-intros*]:  
**fixes**  $x\ d\ k :: nat$  **and**  $tps0 :: tape\ list$   
**assumes**  $k = length\ tps0$   
**and**  $j1 < k\ j2 < k\ j3 < k$   
**assumes**  $j\text{-neq}$  [*simp*]:  $j1 \neq j2\ j2 \neq j3\ j3 \neq j1$   
**and**  $j\text{-gt}$  [*simp*]:  $0 < j2\ 0 < j3\ 0 < j1$   
**and**  $tps0 ! j1 = (\lfloor x \rfloor_N, 1)$   
**and**  $tps0 ! j2 = (\lfloor 0 \rfloor_N, 1)$   
**and**  $tps0 ! j3 = (\lfloor x ^ d \rfloor_N, 1)$   
**and**  $ttt = 71 + 99 * (Suc\ d) ^ 2 * (nlength\ x) ^ 2$   
**and**  $tps' = tps0\ [j3 := (\lfloor x ^ Suc\ d \rfloor_N, 1)]$   
**shows** *transforms*  $tm3\ tps0\ ttt\ tps'$   
 $\langle proof \rangle$

**context**

**fixes**  $x\ k :: nat$  **and**  $tps0 :: tape\ list$   
**assumes**  $jk$ :  $j1 < k\ j2 < k\ j3 < k\ j1 \neq j2\ j2 \neq j3\ j3 \neq j1\ 0 < j2\ 0 < j3\ 0 < j1\ k = length\ tps0$   
**assumes**  $tps0$ :  
 $tps0 ! j1 = (\lfloor x \rfloor_N, 1)$   
 $tps0 ! j2 = (\lfloor 0 \rfloor_N, 1)$   
 $tps0 ! j3 = (\lfloor 0 \rfloor_N, 1)$

**begin**

**lemma**  $tm4$ :  
**fixes**  $d :: nat$   
**assumes**  $tps' = tps0\ [j3 := (\lfloor x ^ d \rfloor_N, 1)]$   
**and**  $ttt = 12 + 71 * d + 99 * d ^ 3 * (nlength\ x) ^ 2$   
**shows** *transforms*  $(tm4\ d)\ tps0\ ttt\ tps'$   
 $\langle proof \rangle$

end

end

**lemma** *transforms-tm-pow* [*transforms-intros*]:

**fixes**  $d :: \text{nat}$

**assumes**  $j1 \neq j2 \ j2 \neq j3 \ j3 \neq j1 \ 0 < j2 \ 0 < j3 \ 0 < j1 \ j1 < k \ j2 < k \ j3 < k \ k = \text{length } tps$

**assumes**

$tps ! j1 = ([x]_N, 1)$

$tps ! j2 = ([0]_N, 1)$

$tps ! j3 = ([0]_N, 1)$

**assumes**  $ttt = 12 + 71 * d + 99 * d^3 * (\text{nlength } x)^2$

**assumes**  $tps' = tps [j3 := ([x^d]_N, 1)]$

**shows** *transforms* (*tm-pow*  $d \ j1 \ j2 \ j3$ )  $tps \ ttt \ tps'$

*<proof>*

### 2.7.7 Monomials

A monomial is a power multiplied by a constant coefficient. The following Turing machines have parameters  $c$  and  $d$  and expect a number  $x$  on tape  $j$ . They output  $c \cdot x^d$  on tape  $j + 3$ . The tapes  $j + 1$  and  $j + 2$  are scratch space for use by *tm-pow* and *tm-mult*.

**definition** *tm-monomial*  $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**

*tm-monomial*  $c \ d \ j \equiv$

*tm-pow*  $d \ j \ (j + 1) \ (j + 2) ;;$

*tm-setn*  $(j + 1) \ c ;;$

*tm-mult*  $(j + 1) \ (j + 2) \ (j + 3);;$

*tm-setn*  $(j + 1) \ 0 ;;$

*tm-setn*  $(j + 2) \ 0$

**lemma** *tm-monomial-tm*:

**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $j + 3 < k$  **and**  $0 < j$

**shows** *turing-machine*  $k \ G \ (\text{tm-monomial } c \ d \ j)$

*<proof>*

**locale** *turing-machine-monomial* =

**fixes**  $c \ d :: \text{nat}$  **and**  $j :: \text{tapeidx}$

**begin**

**definition**  $tm1 \equiv \text{tm-pow } d \ j \ (j + 1) \ (j + 2)$

**definition**  $tm2 \equiv tm1 ;; \text{tm-setn } (j + 1) \ c$

**definition**  $tm3 \equiv tm2 ;; \text{tm-mult } (j + 1) \ (j + 2) \ (j + 3)$

**definition**  $tm4 \equiv tm3 ;; \text{tm-setn } (j + 1) \ 0$

**definition**  $tm5 \equiv tm4 ;; \text{tm-setn } (j + 2) \ 0$

**lemma** *tm5-eq-tm-monomial*:  $tm5 = \text{tm-monomial } c \ d \ j$

*<proof>*

**context**

**fixes**  $x \ k :: \text{nat}$  **and**  $tps0 :: \text{tape list}$

**assumes**  $jk: k = \text{length } tps0 \ j + 3 < k \ 0 < j$

**assumes**  $tps0$ :

$tps0 ! j = ([x]_N, 1)$

$tps0 ! (j + 1) = ([0]_N, 1)$

$tps0 ! (j + 2) = ([0]_N, 1)$

$tps0 ! (j + 3) = ([0]_N, 1)$

**begin**

**definition**  $tps1 \equiv tps0 [(j + 2) := ([x^d]_N, 1)]$

**lemma** *tm1* [*transforms-intros*]:

**assumes**  $ttt = 12 + 71 * d + 99 * d^3 * (\text{nlength } x)^2$

**shows** *transforms*  $tm1 \ tps0 \ ttt \ tps1$

*<proof>*

**definition**  $tps2 \equiv tps0$

$[j + 2 := ([x \hat{d}]_N, 1),$   
 $j + 1 := ([c]_N, 1)]$

**lemma**  $tm2$  [transforms-intros]:

**assumes**  $ttt = 22 + 71 * d + 99 * d \hat{3} * (nlength\ x)^2 + 2 * nlength\ c$

**shows**  $transforms\ tm2\ tps0\ ttt\ tps2$

$\langle proof \rangle$

**definition**  $tps3 \equiv tps0$

$[j + 2 := ([x \hat{d}]_N, 1),$   
 $j + 1 := ([c]_N, 1),$   
 $j + 3 := ([c * x \hat{d}]_N, 1)]$

**lemma**  $tm3$  [transforms-intros]:

**assumes**  $ttt = 26 + 71 * d + 99 * d \hat{3} * (nlength\ x)^2 + 2 * nlength\ c +$   
 $26 * (nlength\ c + nlength\ (x \hat{d})) \hat{2}$

**shows**  $transforms\ tm3\ tps0\ ttt\ tps3$

$\langle proof \rangle$

**definition**  $tps4 \equiv tps0$

$[j + 2 := ([x \hat{d}]_N, 1),$   
 $j + 3 := ([c * x \hat{d}]_N, 1)]$

**lemma**  $tm4$  [transforms-intros]:

**assumes**  $ttt = 36 + 71 * d + 99 * d \hat{3} * (nlength\ x)^2 + 4 * nlength\ c +$   
 $26 * (nlength\ c + nlength\ (x \hat{d})) \hat{2}$

**shows**  $transforms\ tm4\ tps0\ ttt\ tps4$

$\langle proof \rangle$

**definition**  $tps5 \equiv tps0$

$[j + 3 := ([c * x \hat{d}]_N, 1)]$

**lemma**  $tm5$ :

**assumes**  $ttt = 46 + 71 * d + 99 * d \hat{3} * (nlength\ x)^2 + 4 * nlength\ c +$   
 $26 * (nlength\ c + nlength\ (x \hat{d})) \hat{2} +$

$(2 * nlength\ (x \hat{d}))$

**shows**  $transforms\ tm5\ tps0\ ttt\ tps5$

$\langle proof \rangle$

**lemma**  $tm5'$ :

**assumes**  $ttt = 46 + 71 * d + 99 * d \hat{3} * (nlength\ x)^2 + 32 * (nlength\ c + nlength\ (x \hat{d})) \hat{2}$

**shows**  $transforms\ tm5\ tps0\ ttt\ tps5$

$\langle proof \rangle$

**end**

**end**

**lemma**  $transforms-tm-monomialI$  [transforms-intros]:

**fixes**  $ttt\ x\ k :: nat$  **and**  $tps\ tps' :: tape\ list$  **and**  $j :: tapeidx$

**assumes**  $j > 0$  **and**  $j + 3 < k$  **and**  $k = length\ tps$

**assumes**

$tps\ !\ j = ([x]_N, 1)$

$tps\ !\ (j + 1) = ([0]_N, 1)$

$tps\ !\ (j + 2) = ([0]_N, 1)$

$tps\ !\ (j + 3) = ([0]_N, 1)$

**assumes**  $ttt = 46 + 71 * d + 99 * d \hat{3} * (nlength\ x)^2 + 32 * (nlength\ c + nlength\ (x \hat{d})) \hat{2}$

**assumes**  $tps' = tps[j + 3 := ([c * x \hat{d}]_N, 1)]$

**shows**  $transforms\ (tm-monomial\ c\ d\ j)\ tps\ ttt\ tps'$

$\langle proof \rangle$

## 2.7.8 Polynomials

A polynomial is a sum of monomials. In this section we construct for every polynomial function  $p$  a Turing machine that on input  $x \in \mathbb{N}$  outputs  $p(x)$ .

According to our definition of polynomials (see Section 2.1.4), we can represent each polynomial by a list of coefficients. The value of such a polynomial with coefficient list  $cs$  on input  $x$  is given by the next function. In the following definition, the coefficients of the polynomial are in reverse order, which simplifies the Turing machine later.

**definition** *polyvalue* :: *nat list*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**  
*polyvalue*  $cs \equiv (\sum i \leftarrow [0..<length\ cs]. rev\ cs\ !\ i * x \wedge i)$

**lemma** *polyvalue-Nil*: *polyvalue* []  $x = 0$   
*<proof>*

**lemma** *sum-upt-snoc*:  $(\sum i \leftarrow [0..<length\ (zs\ @\ [z])]. (zs\ @\ [z])\ !\ i * x \wedge i) =$   
 $(\sum i \leftarrow [0..<length\ zs]. zs\ !\ i * x \wedge i) + z * x \wedge (length\ zs)$   
*<proof>*

**lemma** *polyvalue-Cons*: *polyvalue*  $(c\ \#\ cs)\ x = c * x \wedge (length\ cs) + polyvalue\ cs\ x$   
*<proof>*

**lemma** *polyvalue-Cons-ge*: *polyvalue*  $(c\ \#\ cs)\ x \geq polyvalue\ cs\ x$   
*<proof>*

**lemma** *polyvalue-Cons-ge2*: *polyvalue*  $(c\ \#\ cs)\ x \geq c * x \wedge (length\ cs)$   
*<proof>*

**lemma** *sum-list-const*:  $(\sum \leftarrow ns.\ c) = c * length\ ns$   
*<proof>*

**lemma** *polyvalue-le*: *polyvalue*  $cs\ x \leq Max\ (set\ cs) * length\ cs * Suc\ x \wedge length\ cs$   
*<proof>*

**lemma** *nlength-polyvalue*:  
 $nlength\ (polyvalue\ cs\ x) \leq nlength\ (Max\ (set\ cs)) + nlength\ (length\ cs) + Suc\ (length\ cs * nlength\ (Suc\ x))$   
*<proof>*

The following Turing machines compute polynomials given as lists of coefficients. If the polynomial is given by coefficients  $cs$ , the TM *tm-polycoef*  $cs\ j$  expect a number  $n$  on tape  $j$  and writes  $p(n)$  to tape  $j + 4$ . The tapes  $j + 1$ ,  $j + 2$ , and  $j + 3$  are auxiliary tapes for use by *tm-monomial*.

**fun** *tm-polycoef* :: *nat list*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**  
*tm-polycoef* []  $j = []$  |  
*tm-polycoef*  $(c\ \#\ cs)\ j =$   
*tm-polycoef*  $cs\ j$  ;;  
 $(tm-monomial\ c\ (length\ cs)\ j$  ;;  
 $tm-add\ (j + 3)\ (j + 4)$  ;;  
 $tm-setn\ (j + 3)\ 0)$

**lemma** *tm-polycoef-tm*:  
**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $j + 4 < k$  **and**  $0 < j$   
**shows** *turing-machine*  $k\ G\ (tm-polycoef\ cs\ j)$   
*<proof>*

**locale** *turing-machine-polycoef* =  
**fixes**  $j ::$  *tapeidx*  
**begin**

**definition** *tm1*  $c\ cs \equiv tm-monomial\ c\ (length\ cs)\ j$

**definition** *tm2*  $c\ cs \equiv tm1\ c\ cs$  ;;  $tm-add\ (j + 3)\ (j + 4)$

**definition** *tm3*  $c\ cs \equiv tm2\ c\ cs$  ;;  $tm-setn\ (j + 3)\ 0$

**fun** *tm4* :: *nat list*  $\Rightarrow$  *machine* **where**

$tm4 [] = []$   
 $tm4 (c \# cs) = tm4 cs ;; tm3 c cs$

**lemma** *tm4-eq-tm-polycoef*:  $tm4 zs = tm\text{-polycoef} zs j$   
 <proof>

**context**

**fixes**  $x y k :: nat$  **and**  $tps0 :: tape\ list$   
**fixes**  $c :: nat$  **and**  $cs :: nat\ list$   
**assumes**  $jk: 0 < j \ j + 4 < k \ k = length\ tps0$   
**assumes**  $tps0$ :  
 $tps0 ! j = ([x]_N, 1)$   
 $tps0 ! (j + 1) = ([0]_N, 1)$   
 $tps0 ! (j + 2) = ([0]_N, 1)$   
 $tps0 ! (j + 3) = ([0]_N, 1)$   
 $tps0 ! (j + 4) = ([y]_N, 1)$

**begin**

**abbreviation**  $d \equiv length\ cs$

**definition**  $tps1 \equiv tps0$   
 $[j + 3 := ([c * x \wedge (length\ cs)]_N, 1)]$

**lemma** *tm1* [*transforms-intros*]:  
**assumes**  $ttt = 46 + 71 * d + 99 * d \wedge 3 * (nlength\ x)^2 +$   
 $32 * (nlength\ c + nlength\ (x \wedge d))^2$   
**shows**  $transforms\ (tm1\ c\ cs)\ tps0\ ttt\ tps1$   
 <proof>

**definition**  $tps2 = tps0$   
 $[j + 3 := ([c * x \wedge (length\ cs)]_N, 1),$   
 $j + 4 := ([c * x \wedge (length\ cs) + y]_N, 1)]$

**lemma** *tm2* [*transforms-intros*]:  
**assumes**  $ttt = 46 + 71 * d + 99 * d \wedge 3 * (nlength\ x)^2 +$   
 $32 * (nlength\ c + nlength\ (x \wedge d))^2 +$   
 $(3 * max\ (nlength\ (c * x \wedge d))\ (nlength\ y) + 10)$   
**shows**  $transforms\ (tm2\ c\ cs)\ tps0\ ttt\ tps2$   
 <proof>

**definition**  $tps3 \equiv tps0$   
 $[j + 4 := ([c * x \wedge d + y]_N, 1)]$

**lemma** *tm3*:  
**assumes**  $ttt = 66 + 71 * d + 99 * d \wedge 3 * (nlength\ x)^2 +$   
 $32 * (nlength\ c + nlength\ (x \wedge d))^2 +$   
 $3 * max\ (nlength\ (c * x \wedge d))\ (nlength\ y) +$   
 $2 * nlength\ (c * x \wedge d)$   
**shows**  $transforms\ (tm3\ c\ cs)\ tps0\ ttt\ tps3$   
 <proof>

**definition**  $tps3' \equiv tps0$   
 $[j + 4 := ([c * x \wedge length\ cs + y]_N, 1)]$

**lemma** *tm3'*:  
**assumes**  $ttt = 66 + 71 * d + 99 * d \wedge 3 * (nlength\ x)^2 +$   
 $32 * (nlength\ c + nlength\ (x \wedge d))^2 +$   
 $5 * max\ (nlength\ (c * x \wedge d))\ (nlength\ y)$   
**shows**  $transforms\ (tm3\ c\ cs)\ tps0\ ttt\ tps3'$   
 <proof>

**end**

**lemma** *tm3''* [*transforms-intros*]:  
**fixes**  $c :: \text{nat}$  **and**  $cs :: \text{nat list}$   
**fixes**  $x k :: \text{nat}$  **and**  $tps0 tps' :: \text{tape list}$   
**assumes**  $k = \text{length } tps0$  **and**  $j + 4 < k$  **and**  $0 < j$   
**assumes**  
 $tps0 ! j = (\lfloor x \rfloor_N, 1)$   
 $tps0 ! (j + 1) = (\lfloor 0 \rfloor_N, 1)$   
 $tps0 ! (j + 2) = (\lfloor 0 \rfloor_N, 1)$   
 $tps0 ! (j + 3) = (\lfloor 0 \rfloor_N, 1)$   
 $tps0 ! (j + 4) = (\lfloor \text{polyvalue } cs \ x \rfloor_N, 1)$   
**assumes**  $ttt = 66 +$   
 $71 * (\text{length } cs) +$   
 $99 * (\text{length } cs) ^ 3 * (\text{nlength } x)^2 +$   
 $32 * (\text{nlength } c + \text{nlength } (x \wedge (\text{length } cs)))^2 +$   
 $5 * \text{max } (\text{nlength } (c * x \wedge (\text{length } cs))) (\text{nlength } (\text{polyvalue } cs \ x))$   
**assumes**  $tps' = tps0$   
 $[j + 4 := (\lfloor \text{polyvalue } (c \# cs) \ x \rfloor_N, 1)]$   
**shows** *transforms* (*tm3*  $c \ cs$ )  $tps0 \ ttt \ tps'$   
 $\langle \text{proof} \rangle$

**lemma** *pow-le-pow-Suc*:  
**fixes**  $a \ b :: \text{nat}$   
**shows**  $a \wedge b \leq \text{Suc } a \wedge \text{Suc } b$   
 $\langle \text{proof} \rangle$

**lemma** *tm4*:  
**fixes**  $x k :: \text{nat}$  **and**  $tps0 :: \text{tape list}$   
**fixes**  $cs :: \text{nat list}$   
**assumes**  $k = \text{length } tps0$  **and**  $j + 4 < k$  **and**  $0 < j$   
**assumes**  
 $tps0 ! j = (\lfloor x \rfloor_N, 1)$   
 $tps0 ! (j + 1) = (\lfloor 0 \rfloor_N, 1)$   
 $tps0 ! (j + 2) = (\lfloor 0 \rfloor_N, 1)$   
 $tps0 ! (j + 3) = (\lfloor 0 \rfloor_N, 1)$   
 $tps0 ! (j + 4) = (\lfloor 0 \rfloor_N, 1)$   
**assumes**  $ttt: ttt = \text{length } cs *$   
 $(66 +$   
 $71 * (\text{length } cs) +$   
 $99 * (\text{length } cs) ^ 3 * (\text{nlength } x)^2 +$   
 $32 * (\text{Max } (\text{set } (\text{map } \text{nlength } cs)) + \text{nlength } (\text{Suc } x \wedge \text{length } cs))^2 +$   
 $5 * \text{nlength } (\text{polyvalue } cs \ x))$   
**shows** *transforms* (*tm4*  $cs$ )  $tps0 \ ttt \ (tps0[j + 4 := (\lfloor \text{polyvalue } cs \ x \rfloor_N, 1)])$   
 $\langle \text{proof} \rangle$

**end**

The time bound in the previous lemma for *tm-polycoef* is a bit unwieldy. It depends not only on the length of the input  $x$  but also on the list of coefficients of the polynomial  $p$  and on the value  $p(x)$ . Next we bound this time bound by a simpler expression of the form  $d + d \cdot |x|^2$  where  $d$  depends only on the polynomial. This is accomplished by the next three lemmas.

**lemma** *tm-polycoef-time-1*:  $\exists d. \forall x. \text{nlength } (\text{polyvalue } cs \ x) \leq d + d * \text{nlength } x$   
 $\langle \text{proof} \rangle$

**lemma** *tm-polycoef-time-2*:  $\exists d. \forall x. (\text{Max } (\text{set } (\text{map } \text{nlength } cs)) + \text{nlength } (\text{Suc } x \wedge \text{length } cs))^2 \leq d + d * \text{nlength } x ^ 2$   
 $\langle \text{proof} \rangle$

**lemma** *tm-polycoef-time-3*:  
 $\exists d. \forall x. \text{length } cs *$   
 $(66 +$   
 $71 * \text{length } cs +$   
 $99 * \text{length } cs ^ 3 * (\text{nlength } x)^2 +$   
 $32 * (\text{Max } (\text{set } (\text{map } \text{nlength } cs)) + \text{nlength } (\text{Suc } x \wedge \text{length } cs))^2 +$

$5 * nlength (polyvalue cs x) \leq d + d * nlength x \wedge 2$   
 (proof)

According to our definition of *polynomial* (see Section 2.1.4) every polynomial has a list of coefficients. Therefore the next definition is well-defined for polynomials  $p$ .

**definition** *coefficients* :: (nat  $\Rightarrow$  nat)  $\Rightarrow$  nat list **where**  
*coefficients*  $p \equiv$  SOME cs.  $\forall n. p n = (\sum i \leftarrow [0..<length cs]. cs ! i * n \wedge i)$

The  $d$  in our upper bound of the form  $d + d \cdot |x|^2$  for the running time of *tm-polycoef* depends on the polynomial. It is given by the next function:

**definition** *d-polynomial* :: (nat  $\Rightarrow$  nat)  $\Rightarrow$  nat **where**  
*d-polynomial*  $p \equiv$   
 (let cs = rev (coefficients p)  
 in SOME d.  $\forall x. length cs * (66 + 71 * length cs + 99 * length cs \wedge 3 * (nlength x)^2 + 32 * (Max (set (map nlength cs)) + nlength (Suc x \wedge length cs))^2 + 5 * nlength (polyvalue cs x)) \leq d + d * nlength x \wedge 2)$

The Turing machine *tm-polycoef* has the coefficients of a polynomial as parameter. Next we devise a similar Turing machine that has the polynomial, as a function  $\mathbb{N} \rightarrow \mathbb{N}$ , as parameter.

**definition** *tm-polynomial* :: (nat  $\Rightarrow$  nat)  $\Rightarrow$  tapeidx  $\Rightarrow$  machine **where**  
*tm-polynomial*  $p j \equiv tm-polycoef (rev (coefficients p)) j$

**lemma** *tm-polynomial-tm*:  
**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j$  **and**  $j + 4 < k$   
**shows** *turing-machine*  $k G (tm-polynomial p j)$   
 (proof)

**lemma** *transforms-tm-polynomialI* [*transforms-intros*]:  
**fixes**  $p :: nat \Rightarrow nat$  **and**  $j :: tapeidx$   
**fixes**  $k x :: nat$  **and**  $tps tps' :: tape list$   
**assumes**  $0 < j$  **and**  $k = length tps$  **and**  $j + 4 < k$   
**and** *polynomial*  $p$   
**assumes**  
 $tps ! j = (\lfloor x \rfloor_N, 1)$   
 $tps ! (j + 1) = (\lfloor 0 \rfloor_N, 1)$   
 $tps ! (j + 2) = (\lfloor 0 \rfloor_N, 1)$   
 $tps ! (j + 3) = (\lfloor 0 \rfloor_N, 1)$   
 $tps ! (j + 4) = (\lfloor 0 \rfloor_N, 1)$   
**assumes**  $ttt = d-polynomial p + d-polynomial p * nlength x \wedge 2$   
**assumes**  $tps' = tps$   
 $[j + 4 := (\lfloor p x \rfloor_N, 1)]$   
**shows** *transforms*  $(tm-polynomial p j) tps ttt tps'$   
 (proof)

## 2.7.9 Division by two

In order to divide a number by two, a Turing machine can shift all symbols on the tape containing the number to the left, of course without overwriting the start symbol.

The next command implements the left shift. It scans the tape  $j$  from right to left and memorizes the current symbol on the last tape. It works very similar to *cmd-double* only in the opposite direction. Upon reaching the start symbol, it moves the head one cell to the right.

**definition** *cmd-halve* :: tapeidx  $\Rightarrow$  command **where**  
*cmd-halve*  $j rs \equiv$   
 (if  $rs ! j = 1$  then 1 else 0,  
 (map ( $\lambda i.$   
 if  $i = j$  then  
 if  $rs ! j = \triangleright$  then  $(rs ! i, Right)$   
 else if last  $rs = \triangleright$  then  $(\square, Left)$

else (tosym (todigit (last rs)), Left)  
 else if  $i = \text{length } rs - 1$  then (tosym (todigit (rs ! j)), Stay)  
 else (rs ! i, Stay) [0..<length rs])

**lemma** *turing-command-halve*:

assumes  $G \geq 4$  and  $0 < j$  and  $j < k$   
 shows *turing-command* (Suc k) 1 G (cmd-halve j)  
 ⟨proof⟩

**lemma** *sem-cmd-halve-2*:

assumes  $j < k$   
 and bit-symbols  $xs$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $i \leq \text{length } xs$   
 and  $i > 0$   
 and  $z = \mathbf{0} \vee z = \mathbf{1}$   
 and  $tps ! j = (\lfloor xs \rfloor, i)$   
 and  $tps ! k = \lceil z \rceil$   
 and  $tps' = tps[j := tps ! j \mid := z \mid - 1, k := \lceil xs ! (i - 1) \rceil]$   
 shows *sem* (cmd-halve j) (0, tps) = (0, tps')  
 ⟨proof⟩

**lemma** *sem-cmd-halve-1*:

assumes  $j < k$   
 and bit-symbols  $xs$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $0 < \text{length } xs$   
 and  $tps ! j = (\lfloor xs \rfloor, \text{length } xs)$   
 and  $tps ! k = \lceil \triangleright \rceil$   
 and  $tps' = tps[j := tps ! j \mid := \square \mid - 1, k := \lceil xs ! (\text{length } xs - 1) \rceil]$   
 shows *sem* (cmd-halve j) (0, tps) = (0, tps')  
 ⟨proof⟩

**lemma** *sem-cmd-halve-0*:

assumes  $j < k$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $tps ! j = (\lfloor xs \rfloor, 0)$   
 and  $tps ! k = \lceil z \rceil$   
 and  $tps' = tps[j := tps ! j \mid + 1, k := \lceil \mathbf{0} \rceil]$   
 shows *sem* (cmd-halve j) (0, tps) = (1, tps')  
 ⟨proof⟩

**definition** *tm-halve* :: *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-halve* j  $\equiv$  [cmd-halve j]

**lemma** *tm-halve-tm*:

assumes  $k \geq 2$  and  $G \geq 4$  and  $0 < j$  and  $j < k$   
 shows *turing-machine* (Suc k) G (tm-halve j)  
 ⟨proof⟩

**lemma** *exe-cmd-halve-0*:

assumes  $j < k$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $tps ! j = (\lfloor xs \rfloor, 0)$   
 and  $tps ! k = \lceil z \rceil$   
 and  $tps' = tps[j := tps ! j \mid + 1, k := \lceil \mathbf{0} \rceil]$   
 shows *exe* (tm-halve j) (0, tps) = (1, tps')  
 ⟨proof⟩

**lemma** *execute-cmd-halve-0*:

assumes  $j < k$   
 and  $\text{length } tps = \text{Suc } k$   
 and  $tps ! j = (\lfloor \square \rfloor, 0)$

**and**  $tps ! k = \lceil \triangleright \rceil$   
**and**  $tps' = tps[j := tps ! j \mid + \mid 1, k := \lceil \mathbf{0} \rceil]$   
**shows**  $execute (tm-halve j) (0, tps) 1 = (1, tps')$   
 $\langle proof \rangle$

**definition**  $shift :: tape \Rightarrow nat \Rightarrow tape$  **where**  
 $shift\ tp\ y \equiv (\lambda x. \text{if } x \leq y \text{ then } (fst\ tp)\ x \text{ else } (fst\ tp)\ (Suc\ x), y)$

**lemma**  $shift\text{-update}: y > 0 \implies shift\ tp\ y \mid := \mid (fst\ tp)\ (Suc\ y) \mid - \mid 1 = shift\ tp\ (y - 1)$   
 $\langle proof \rangle$

**lemma**  $shift\text{-contents-0}$ :  
**assumes**  $length\ xs > 0$   
**shows**  $shift\ (\lfloor xs \rfloor, length\ xs)\ 0 = (\lfloor tl\ xs \rfloor, 0)$   
 $\langle proof \rangle$

**lemma**  $proper\text{-bit-symbols}$ :  $bit\text{-symbols}\ ws \implies proper\text{-symbols}\ ws$   
 $\langle proof \rangle$

**lemma**  $bit\text{-symbols-shift}$ :  
**assumes**  $t < length\ ws$  **and**  $bit\text{-symbols}\ ws$   
**shows**  $\mid \cdot \mid (shift\ (\lfloor ws \rfloor, length\ ws)\ (length\ ws - t)) \neq 1$   
 $\langle proof \rangle$

**lemma**  $exe\text{-cmd-halve-1}$ :  
**assumes**  $j < k$   
**and**  $length\ tps = Suc\ k$   
**and**  $bit\text{-symbols}\ xs$   
**and**  $length\ xs > 0$   
**and**  $tps ! j = (\lfloor xs \rfloor, length\ xs)$   
**and**  $tps ! k = \lceil \triangleright \rceil$   
**and**  $tps' = tps[j := tps ! j \mid := \mid \square \mid - \mid 1, k := \lceil xs ! (length\ xs - 1) \rceil]$   
**shows**  $exe (tm-halve j) (0, tps) = (0, tps')$   
 $\langle proof \rangle$

**lemma**  $shift\text{-contents-eq-take-drop}$ :  
**assumes**  $length\ xs > 0$   
**and**  $ys = take\ i\ xs @ drop\ (Suc\ i)\ xs$   
**and**  $i > 0$   
**and**  $i < length\ xs$   
**shows**  $shift\ (\lfloor xs \rfloor, length\ xs)\ i = (\lfloor ys \rfloor, i)$   
 $\langle proof \rangle$

**lemma**  $exe\text{-cmd-halve-2}$ :  
**assumes**  $j < k$   
**and**  $bit\text{-symbols}\ xs$   
**and**  $length\ tps = Suc\ k$   
**and**  $i \leq length\ xs$   
**and**  $i > 0$   
**and**  $z = \mathbf{0} \vee z = \mathbf{1}$   
**and**  $tps ! j = (\lfloor xs \rfloor, i)$   
**and**  $tps ! k = \lceil z \rceil$   
**and**  $tps' = tps[j := tps ! j \mid := \mid z \mid - \mid 1, k := \lceil xs ! (i - 1) \rceil]$   
**shows**  $exe (tm-halve j) (0, tps) = (0, tps')$   
 $\langle proof \rangle$

**lemma**  $shift\text{-contents-length-minus-1}$ :  
**assumes**  $length\ xs > 0$   
**shows**  $shift\ (\lfloor xs \rfloor, length\ xs)\ (length\ xs - 1) = (\lfloor xs \rfloor, length\ xs) \mid := \mid \square \mid - \mid 1$   
 $\langle proof \rangle$

**lemma**  $execute\text{-tm-halve-1-less}$ :  
**assumes**  $j < k$

**and**  $\text{length } tps = \text{Suc } k$   
**and**  $\text{bit-symbols } xs$   
**and**  $\text{length } xs > 0$   
**and**  $tps ! j = (\lfloor xs \rfloor, \text{length } xs)$   
**and**  $tps ! k = \lceil \triangleright \rceil$   
**and**  $t \geq 1$   
**and**  $t \leq \text{length } xs$   
**shows**  $\text{execute } (tm\text{-halve } j) (0, tps) t = (0, tps$   
 $\quad [j := \text{shift } (tps ! j) (\text{length } xs - t),$   
 $\quad k := \lceil xs ! (\text{length } xs - t) \rceil])$   
 $\langle \text{proof} \rangle$

**lemma** *execute-tm-halve-1*:

**assumes**  $j < k$   
**and**  $\text{length } tps = \text{Suc } k$   
**and**  $\text{bit-symbols } xs$   
**and**  $\text{length } xs > 0$   
**and**  $tps ! j = (\lfloor xs \rfloor, \text{length } xs)$   
**and**  $tps ! k = \lceil \triangleright \rceil$   
**and**  $tps[j := (\lfloor tl \ xs \rfloor, 1), k := \lceil \mathbf{0} \rceil]$   
**shows**  $\text{execute } (tm\text{-halve } j) (0, tps) (\text{Suc } (\text{length } xs)) = (1, tps')$   
 $\langle \text{proof} \rangle$

**lemma** *execute-tm-halve*:

**assumes**  $j < k$   
**and**  $\text{length } tps = \text{Suc } k$   
**and**  $\text{bit-symbols } xs$   
**and**  $tps ! j = (\lfloor xs \rfloor, \text{length } xs)$   
**and**  $tps ! k = \lceil \triangleright \rceil$   
**and**  $tps' = tps[j := (\lfloor tl \ xs \rfloor, 1), k := \lceil \mathbf{0} \rceil]$   
**shows**  $\text{execute } (tm\text{-halve } j) (0, tps) (\text{Suc } (\text{length } xs)) = (1, tps')$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-halve*:

**assumes**  $j < k$   
**and**  $\text{length } tps = \text{Suc } k$   
**and**  $\text{bit-symbols } xs$   
**and**  $tps ! j = (\lfloor xs \rfloor, \text{length } xs)$   
**and**  $tps ! k = \lceil \triangleright \rceil$   
**and**  $tps' = tps[j := (\lfloor tl \ xs \rfloor, 1), k := \lceil \mathbf{0} \rceil]$   
**shows**  $\text{transforms } (tm\text{-halve } j) tps (\text{Suc } (\text{length } xs)) tps'$   
 $\langle \text{proof} \rangle$

**lemma** *transforms-tm-halve2*:

**assumes**  $j < k$   
**and**  $\text{length } tps = k$   
**and**  $\text{bit-symbols } xs$   
**and**  $tps ! j = (\lfloor xs \rfloor, \text{length } xs)$   
**and**  $tps' = tps[j := (\lfloor tl \ xs \rfloor, 1)]$   
**shows**  $\text{transforms } (tm\text{-halve } j) (tps @ \lceil \lceil \triangleright \rceil \rceil) (\text{Suc } (\text{length } xs)) (tps' @ \lceil \lceil \mathbf{0} \rceil \rceil)$   
 $\langle \text{proof} \rangle$

The next Turing machine removes the memorization tape from *tm-halve*.

**definition** *tm-halve'* :: *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-halve' j*  $\equiv$  *cartesian* (*tm-halve j*) 4

**lemma** *bounded-write-tm-halve*:

**assumes**  $j < k$   
**shows**  $\text{bounded-write } (tm\text{-halve } j) k 4$   
 $\langle \text{proof} \rangle$

**lemma** *immobile-tm-halve*:

**assumes**  $j < k$

**shows** *immobile* (*tm-halve* *j*) *k* (*Suc* *k*)  
 ⟨*proof*⟩

**lemma** *tm-halve'-tm*:

**assumes**  $G \geq 4$  **and**  $0 < j$  **and**  $j < k$   
**shows** *turing-machine* *k* *G* (*tm-halve'* *j*)  
 ⟨*proof*⟩

**lemma** *transforms-tm-halve'* [*transforms-intros*]:

**assumes**  $j > 0$  **and**  $j < k$   
**and** *length* *tps* = *k*  
**and** *bit-symbols* *xs*  
**and**  $tps ! j = (\lfloor xs \rfloor, \text{length } xs)$   
**and**  $tps' = tps[j := (\lfloor tl \ xs \rfloor, 1)]$   
**shows** *transforms* (*tm-halve'* *j*) *tps* (*Suc* (*length* *xs*)) *tps'*  
 ⟨*proof*⟩

**lemma** *num-tl-div-2*:  $\text{num } (tl \ xs) = \text{num } xs \ \text{div } 2$   
 ⟨*proof*⟩

**lemma** *canrepr-div-2*:  $\text{canrepr } (n \ \text{div } 2) = tl \ (\text{canrepr } n)$   
 ⟨*proof*⟩

**corollary** *nlength-times2*:  $nlength \ (2 * n) \leq Suc \ (nlength \ n)$   
 ⟨*proof*⟩

**corollary** *nlength-times2plus1*:  $nlength \ (2 * n + 1) \leq Suc \ (nlength \ n)$   
 ⟨*proof*⟩

The next Turing machine is the one we actually use to divide a number by two. First it moves to the end of the symbol sequence representing the number, then it applies *tm-halve'*.

**definition** *tm-div2* :: *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-div2* *j*  $\equiv$  *tm-right-until* *j*  $\{\square\}$  ;; *tm-left* *j* ;; *tm-halve'* *j*

**lemma** *tm-div2-tm*:

**assumes**  $G \geq 4$  **and**  $0 < j$  **and**  $j < k$   
**shows** *turing-machine* *k* *G* (*tm-div2* *j*)  
 ⟨*proof*⟩

**locale** *turing-machine-div2* =

**fixes** *j* :: *tapeidx*

**begin**

**definition** *tm1*  $\equiv$  *tm-right-until* *j*  $\{\square\}$

**definition** *tm2*  $\equiv$  *tm1* ;; *tm-left* *j*

**definition** *tm3*  $\equiv$  *tm2* ;; *tm-halve'* *j*

**lemma** *tm3-eq-tm-div2*:  $tm3 = tm-div2 \ j$   
 ⟨*proof*⟩

**context**

**fixes** *tps0* :: *tape list* **and** *k* *n* :: *nat*

**assumes** *jk*:  $0 < j \ j < k \ \text{length } tps0 = k$

**and** *tps0*:  $tps0 ! j = (\lfloor n \rfloor_N, 1)$

**begin**

**definition** *tps1*  $\equiv$  *tps0*

$[j := (\lfloor n \rfloor_N, Suc \ (nlength \ n))]$

**lemma** *tm1* [*transforms-intros*]:

**assumes** *ttt* = *Suc* (*nlength* *n*)

**shows** *transforms* *tm1* *tps0* *ttt* *tps1*

⟨*proof*⟩

**definition**  $tps2 \equiv tps0$   
 $[j := (\lfloor n \rfloor_N, nlength\ n)]$

**lemma**  $tm2$  [*transforms-intros*]:  
**assumes**  $ttt = 2 + nlength\ n$   
**shows** *transforms*  $tm2\ tps0\ ttt\ tps2$   
 $\langle proof \rangle$

**definition**  $tps3 \equiv tps0$   
 $[j := (\lfloor n\ div\ 2 \rfloor_N, 1)]$

**lemma**  $tm3$ :  
**assumes**  $ttt = 2 * nlength\ n + 3$   
**shows** *transforms*  $tm3\ tps0\ ttt\ tps3$   
 $\langle proof \rangle$

**end**

**end**

**lemma** *transforms-tm-div2I* [*transforms-intros*]:  
**fixes**  $tps\ tps' :: tape\ list$  **and**  $ttt\ k\ n :: nat$  **and**  $j :: tapeidx$   
**assumes**  $0 < j\ j < k$   
**and**  $length\ tps = k$   
**and**  $tps\ !\ j = (\lfloor n \rfloor_N, 1)$   
**assumes**  $ttt = 2 * nlength\ n + 3$   
**assumes**  $tps' = tps[j := (\lfloor n\ div\ 2 \rfloor_N, 1)]$   
**shows** *transforms*  $(tm-div2\ j)\ tps\ ttt\ tps'$   
 $\langle proof \rangle$

## 2.7.10 Modulo two

In this section we construct a Turing machine that writes to tape  $j_2$  the symbol **1** or  $\square$  depending on whether the number on tape  $j_1$  is odd or even. If initially tape  $j_2$  contained at most one symbol, it will contain the numbers 1 or 0.

**lemma** *canrepr-odd*:  $odd\ n \implies canrepr\ n\ !\ 0 = \mathbf{1}$   
 $\langle proof \rangle$

**lemma** *canrepr-even*:  $even\ n \implies 0 < n \implies canrepr\ n\ !\ 0 = \mathbf{0}$   
 $\langle proof \rangle$

**definition**  $tm-mod2\ j1\ j2 \equiv tm-trans2\ j1\ j2$  ( $\lambda z. if\ z = \mathbf{1}$  then  $\mathbf{1}$  else  $\square$ )

**lemma**  $tm-mod2-tm$ :  
**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$   
**shows** *turing-machine*  $k\ G\ (tm-mod2\ j1\ j2)$   
 $\langle proof \rangle$

**lemma** *transforms-tm-mod2I* [*transforms-intros*]:  
**assumes**  $j1 < length\ tps$  **and**  $0 < j2$  **and**  $j2 < length\ tps$   
**and**  $b \leq 1$   
**assumes**  $tps\ !\ j1 = (\lfloor n \rfloor_N, 1)$   
**and**  $tps\ !\ j2 = (\lfloor b \rfloor_N, 1)$   
**assumes**  $tps' = tps[j2 := (\lfloor n\ mod\ 2 \rfloor_N, 1)]$   
**shows** *transforms*  $(tm-mod2\ j1\ j2)\ tps\ 1\ tps'$   
 $\langle proof \rangle$

## 2.7.11 Boolean operations

In order to support Boolean operations, we represent the value True by the number 1 and False by 0.

**abbreviation**  $bcontents :: bool \Rightarrow (nat \Rightarrow symbol)$  ( $\langle \_ \rfloor_B \rangle$ ) **where**

$\lfloor b \rfloor_B \equiv \lfloor \text{if } b \text{ then } 1 \text{ else } 0 \rfloor_N$

A tape containing a number contains the number 0 iff. there is a blank in cell number 1.

**lemma** *read-ncontents-eq-0*:

**assumes**  $tps ! j = (\lfloor n \rfloor_N, 1)$  **and**  $j < \text{length } tps$   
**shows**  $(\text{read } tps) ! j = \square \longleftrightarrow n = 0$   
*<proof>*

## And

The next Turing machine, when given two numbers  $a, b \in \{0, 1\}$  on tapes  $j_1$  and  $j_2$ , writes to tape  $j_1$  the number 1 if  $a = b = 1$ ; otherwise it writes the number 0. In other words, it overwrites tape  $j_1$  with the logical AND of the two tapes.

**definition** *tm-and* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-and*  $j_1 j_2 \equiv \text{IF } \lambda rs. rs ! j_1 = 1 \wedge rs ! j_2 = \square \text{ THEN } \text{tm-write } j_1 \ \square \ \text{ELSE } \square \ \text{ENDIF}$

**lemma** *tm-and-tm*:

**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j_1$  **and**  $j_1 < k$   
**shows** *turing-machine*  $k \ G \ (\text{tm-and } j_1 j_2)$   
*<proof>*

**locale** *turing-machine-and* =

**fixes**  $j_1 j_2 :: \text{tapeidx}$

**begin**

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $k :: \text{nat}$  **and**  $a \ b :: \text{nat}$   
**assumes**  $ab: a < 2 \ b < 2$   
**assumes**  $jk: j_1 < k \ j_2 < k \ j_1 \neq j_2 \ 0 < j_1 \ \text{length } tps0 = k$   
**assumes**  $tps0$ :  
 $tps0 ! j_1 = (\lfloor a \rfloor_N, 1)$   
 $tps0 ! j_2 = (\lfloor b \rfloor_N, 1)$

**begin**

**definition**  $tps1 \equiv tps0$

$[j_1 := (\lfloor a = 1 \wedge b = 1 \rfloor_B, 1)]$

**lemma** *tm*: *transforms*  $(\text{tm-and } j_1 j_2) \ tps0 \ 3 \ tps1$

*<proof>*

**end**

**end**

**lemma** *transforms-tm-andI* [*transforms-intros*]:

**fixes**  $j_1 j_2 :: \text{tapeidx}$   
**fixes**  $tps :: \text{tape list}$  **and**  $k :: \text{nat}$  **and**  $a \ b :: \text{nat}$   
**assumes**  $a < 2 \ b < 2$   
**assumes**  $\text{length } tps = k$   
**assumes**  $j_1 < k \ j_2 < k \ j_1 \neq j_2 \ 0 < j_1$   
**assumes**  
 $tps ! j_1 = (\lfloor a \rfloor_N, 1)$   
 $tps ! j_2 = (\lfloor b \rfloor_N, 1)$   
**assumes**  $tps' = tps$   
 $[j_1 := (\lfloor a = 1 \wedge b = 1 \rfloor_B, 1)]$   
**shows** *transforms*  $(\text{tm-and } j_1 j_2) \ tps \ 3 \ tps'$   
*<proof>*

## Not

The next Turing machine turns the number 1 into 0 and vice versa.

**definition** *tm-not* :: *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-not*  $j \equiv \text{IF } \lambda rs. rs ! j = \square \text{ THEN } \text{tm-write } j \ \mathbf{1} \ \text{ELSE } \text{tm-write } j \ \square \ \text{ENDIF}$

**lemma** *tm-not-tm*:

**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j$  **and**  $j < k$   
**shows** *turing-machine*  $k \ G$  (*tm-not*  $j$ )  
 $\langle \text{proof} \rangle$

**locale** *turing-machine-not* =

**fixes**  $j :: \text{tapeidx}$

**begin**

**context**

**fixes**  $\text{tps0} :: \text{tape list}$  **and**  $k :: \text{nat}$  **and**  $a :: \text{nat}$   
**assumes**  $a < 2$   
**assumes**  $jk: j < k \ \text{length } \text{tps0} = k$   
**assumes**  $\text{tps0}: \text{tps0} ! j = (\lfloor a \rfloor_N, 1)$

**begin**

**definition**  $\text{tps1} \equiv \text{tps0}$

$[j := (\lfloor a \neq 1 \rfloor_B, 1)]$

**lemma** *tm: transforms (tm-not j) tps0 3 tps1*

$\langle \text{proof} \rangle$

**end**

**end**

**lemma** *transforms-tm-notI* [*transforms-intros*]:

**fixes**  $j :: \text{tapeidx}$

**fixes**  $\text{tps} \ \text{tps}' :: \text{tape list}$  **and**  $k :: \text{nat}$  **and**  $a :: \text{nat}$

**assumes**  $j < k \ \text{length } \text{tps} = k$

**and**  $a < 2$

**assumes**  $\text{tps} ! j = (\lfloor a \rfloor_N, 1)$

**assumes**  $\text{tps}' = \text{tps}$

$[j := (\lfloor a \neq 1 \rfloor_B, 1)]$

**shows** *transforms (tm-not j) tps 3 tps'*

$\langle \text{proof} \rangle$

**end**

## 2.8 Lists of numbers

**theory** *Lists-Lists*

**imports** *Arithmetic*

**begin**

In the previous section we defined a representation for numbers over the binary alphabet  $\{0,1\}$ . In this section we first introduce a representation of lists of numbers as symbol sequences over the alphabet  $\{0,1,\lfloor\}$ . Then we define Turing machines for some operations over such lists.

As with the arithmetic operations, Turing machines implementing the operations on lists usually expect the tape heads of the operands to be in position 1 and guarantee to place the tape heads of the result in position 1. The exception are Turing machines for iterating over lists; such TMs move the tape head to the next list element.

A tape containing such representations corresponds to a variable of type *nat list*. A tape in the start configuration corresponds to the empty list of numbers.

### 2.8.1 Representation as symbol sequence

The obvious idea for representing a list of numbers is to write them one after another separated by a fresh symbol, such as  $\lfloor$ . However since we represent the number 0 by the empty symbol sequence,

this would result in both the empty list and the list containing only the number 0 to be represented by the same symbol sequence, namely the empty one. To prevent this we use the symbol | not as a separator but as a terminator; that is, we append it to every number. Consequently the empty symbol sequence represents the empty list, whereas the list [0] is represented by the symbol sequence |. As another example, the list [14, 0, 0, 7] is represented by **0111|||111|**. As a side effect of using terminators instead of separators, the representation of the concatenation of lists is just the concatenation of the representations of the individual lists. Moreover the length of the representation is simply the sum of the individual representation lengths. The number of | symbols equals the number of elements in the list.

This is how lists of numbers are represented as symbol sequences:

**definition** *numlist* :: *nat list*  $\Rightarrow$  *symbol list* **where**  
*numlist* *ns*  $\equiv$  *concat* (*map* ( $\lambda n$ . *canrepr* *n* @ []) *ns*)

**lemma** *numlist-Nil*: *numlist* [] = []  
 <proof>

**proposition** *numlist* [0] = []  
 <proof>

**lemma** *numlist-234*: *set* (*numlist* *ns*)  $\subseteq$  {0, 1, |}  
 <proof>

**lemma** *symbols-lt-numlist*: *symbols-lt* 5 (*numlist* *ns*)  
 <proof>

**lemma** *bit-symbols-prefix-eq*:  
**assumes** (*x* @ []) @ *xs* = (*y* @ []) @ *ys* **and** *bit-symbols* *x* **and** *bit-symbols* *y*  
**shows** *x* = *y*  
 <proof>

**lemma** *numlist-inj*: *numlist* *ns1* = *numlist* *ns2*  $\implies$  *ns1* = *ns2*  
 <proof>

**corollary** *proper-symbols-numlist*: *proper-symbols* (*numlist* *ns*)  
 <proof>

The next property would not hold if we used separators between elements instead of terminators after elements.

**lemma** *numlist-append*: *numlist* (*xs* @ *ys*) = *numlist* *xs* @ *numlist* *ys*  
 <proof>

Like *nlength* for numbers, we have *nllength* for the length of the representation of a list of numbers.

**definition** *nllength* :: *nat list*  $\Rightarrow$  *nat* **where**  
*nllength* *ns*  $\equiv$  *length* (*numlist* *ns*)

**lemma** *nllength*: *nllength* *ns* = ( $\sum n \leftarrow ns$ . *Suc* (*nlength* *n*))  
 <proof>

**lemma** *nllength-Nil* [*simp*]: *nllength* [] = 0  
 <proof>

**lemma** *length-le-nllength*: *length* *ns*  $\leq$  *nllength* *ns*  
 <proof>

**lemma** *nllength-le-len-mult-max*:  
**fixes** *N* :: *nat* **and** *ns* :: *nat list*  
**assumes**  $\forall n \in set\ ns. n \leq N$   
**shows** *nllength* *ns*  $\leq$  *Suc* (*nlength* *N*) \* *length* *ns*  
 <proof>

**lemma** *nllength-upt-le-len-mult-max*:  
**fixes** *a* *b* :: *nat*

**shows**  $nlength [a..<b] \leq Suc (nlength b) * (b - a)$   
 ⟨proof⟩

**lemma** *nlength-le-len-mult-Max*:  $nlength ns \leq Suc (nlength (Max (set ns))) * length ns$   
 ⟨proof⟩

**lemma** *member-le-nlength*:  $n \in set ns \implies nlength n \leq nlength ns$   
 ⟨proof⟩

**lemma** *member-le-nlength-1*:  $n \in set ns \implies nlength n \leq nlength ns - 1$   
 ⟨proof⟩

**lemma** *nlength-gr-0*:  $ns \neq [] \implies 0 < nlength ns$   
 ⟨proof⟩

**lemma** *nlength-min-le-nlength*:  $n \in set ns \implies m \in set ns \implies nlength (min n m) \leq nlength ns$   
 ⟨proof⟩

**lemma** *take-drop-numlist*:  
**assumes**  $i < length ns$   
**shows**  $take (Suc (nlength (ns ! i))) (drop (nlength (take i ns)) (numlist ns)) = canrepr (ns ! i) @ []$   
 ⟨proof⟩

**corollary** *take-drop-numlist'*:  
**assumes**  $i < length ns$   
**shows**  $take (nlength (ns ! i)) (drop (nlength (take i ns)) (numlist ns)) = canrepr (ns ! i)$   
 ⟨proof⟩

**corollary** *numlist-take-at-term*:  
**assumes**  $i < length ns$   
**shows**  $numlist ns ! (nlength (take i ns) + nlength (ns ! i)) = |$   
 ⟨proof⟩

**lemma** *numlist-take-at*:  
**assumes**  $i < length ns$  **and**  $j < nlength (ns ! i)$   
**shows**  $numlist ns ! (nlength (take i ns) + j) = canrepr (ns ! i) ! j$   
 ⟨proof⟩

**lemma** *nlength-take-Suc*:  
**assumes**  $i < length ns$   
**shows**  $nlength (take i ns) + Suc (nlength (ns ! i)) = nlength (take (Suc i) ns)$   
 ⟨proof⟩

**lemma** *numlist-take-Suc-at-term*:  
**assumes**  $i < length ns$   
**shows**  $numlist ns ! (nlength (take (Suc i) ns) - 1) = |$   
 ⟨proof⟩

**lemma** *nlength-take*:  
**assumes**  $i < length ns$   
**shows**  $nlength (take i ns) + nlength (ns ! i) < nlength ns$   
 ⟨proof⟩

The contents of a tape starting with the start symbol  $\triangleright$  followed by the symbol sequence representing a list of numbers:

**definition** *nlcontents* ::  $nat list \Rightarrow (nat \Rightarrow symbol) (\langle \_ \rangle_{NL})$  **where**  
 $[ns]_{NL} \equiv [numlist ns]$

**lemma** *clean-tape-nlcontents*:  $clean-tape ([ns]_{NL}, i)$   
 ⟨proof⟩

**lemma** *nlcontents-Nil*:  $[[]]_{NL} = []$   
 ⟨proof⟩

**lemma** *nlcontents-rneigh-4*:  
**assumes**  $i < \text{length } ns$   
**shows**  $\text{rneigh } ([ns]_{NL}, \text{Suc } (\text{nlength } (\text{take } i \text{ } ns))) \{\}\ (\text{nlength } (ns ! i))$   
 $\langle \text{proof} \rangle$

**lemma** *nlcontents-rneigh-04*:  
**assumes**  $i < \text{length } ns$   
**shows**  $\text{rneigh } ([ns]_{NL}, \text{Suc } (\text{nlength } (\text{take } i \text{ } ns))) \{\square, |\} (\text{nlength } (ns ! i))$   
 $\langle \text{proof} \rangle$

A tape storing a list of numbers, with the tape head in the first blank cell after the symbols:

**abbreviation** *ntlape* :: *nat list*  $\Rightarrow$  *tape* **where**  
 $\text{ntlape } ns \equiv ([ns]_{NL}, \text{Suc } (\text{nlength } ns))$

A tape storing a list of numbers, with the tape head on the first symbol representing the  $i$ -th number, unless the  $i$ -th number is zero, in which case the tape head is on the terminator symbol of this zero. If  $i$  is out of bounds of the list, the tape head is at the first blank after the list.

**abbreviation** *ntlape'* :: *nat list*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape* **where**  
 $\text{ntlape}' \text{ } ns \ i \equiv ([ns]_{NL}, \text{Suc } (\text{nlength } (\text{take } i \text{ } ns)))$

**lemma** *ntlape'-tape-read*:  $|\cdot| (\text{ntlape}' \text{ } ns \ i) = \square \iff i \geq \text{length } ns$   
 $\langle \text{proof} \rangle$

## 2.8.2 Moving to the next element

The next Turing machine provides a means to iterate over a list of numbers. If the TM starts in a configuration where the tape  $j_1$  contains a list of numbers and the tape head is on the first symbol of the  $i$ -th element of this list, then after the TM has finished, the  $i$ -th element will be written on tape  $j_2$  and the tape head on  $j_1$  will have advanced by one list element. If  $i$  is the last element of the list, the tape head on  $j_1$  will be on a blank symbol. One can execute this TM in a loop until the tape head reaches a blank. The TM is generic over a parameter  $z$  representing the terminator symbol, so it can be used for other kinds of lists, too (see Section 2.9).

**definition** *tm-nextract* :: *symbol*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**  
 $\text{tm-nextract } z \ j1 \ j2 \equiv$   
 $\text{tm-erase-cr } j2 \ ;;$   
 $\text{tm-cp-until } j1 \ j2 \ \{z\} \ ;;$   
 $\text{tm-cr } j2 \ ;;$   
 $\text{tm-right } j1$

**lemma** *tm-nextract-tm*:  
**assumes**  $G \geq 4$  **and**  $G > z$  **and**  $0 < j2$  **and**  $j2 < k$  **and**  $j1 < k$  **and**  $k \geq 2$   
**shows**  $\text{turing-machine } k \ G \ (\text{tm-nextract } z \ j1 \ j2)$   
 $\langle \text{proof} \rangle$

The next locale is for the case  $z = |\cdot|$ .

**locale** *turing-machine-nextract-4* =  
**fixes**  $j1 \ j2 :: \text{tapeidx}$   
**begin**

**definition**  $\text{tm1} \equiv \text{tm-erase-cr } j2$   
**definition**  $\text{tm2} \equiv \text{tm1} \ ;; \text{tm-cp-until } j1 \ j2 \ \{\}$   
**definition**  $\text{tm3} \equiv \text{tm2} \ ;; \text{tm-cr } j2$   
**definition**  $\text{tm4} \equiv \text{tm3} \ ;; \text{tm-right } j1$

**lemma** *tm4-eq-tm-nextract*:  $\text{tm4} = \text{tm-nextract } |\cdot| \ j1 \ j2$   
 $\langle \text{proof} \rangle$

**context**

**fixes**  $tps0$  :: *tape list* **and**  $k$   $idx$   $dummy$  :: *nat* **and**  $ns$  :: *nat list*  
**assumes**  $jk$ :  $j1 < k$   $j2 < k$   $0 < j1$   $0 < j2$   $j1 \neq j2$   $length\ tps0 = k$   
**and**  $idx$ :  $idx < length\ ns$   
**and**  $tps0$ :  
 $tps0 ! j1 = nltape' ns\ idx$   
 $tps0 ! j2 = (\lfloor dummy \rfloor_N, 1)$   
**begin**

**definition**  $tps1 \equiv tps0[j2 := (\lfloor 0 \rfloor_N, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:  
**assumes**  $ttt = 7 + 2 * nlength\ dummy$   
**shows** *transforms*  $tm1\ tps0\ ttt\ tps1$   
*<proof>*

**definition**  $tps2 \equiv tps0$   
 $[j1 := (\lfloor ns \rfloor_{NL}, nlength\ (take\ (Suc\ idx)\ ns)),$   
 $j2 := (\lfloor ns ! idx \rfloor_N, Suc\ (nlength\ (ns ! idx)))]$

**lemma**  $tm2$  [*transforms-intros*]:  
**assumes**  $ttt = 7 + 2 * nlength\ dummy + Suc\ (nlength\ (ns ! idx))$   
**shows** *transforms*  $tm2\ tps0\ ttt\ tps2$   
*<proof>*

**definition**  $tps3 \equiv tps0$   
 $[j1 := (\lfloor ns \rfloor_{NL}, nlength\ (take\ (Suc\ idx)\ ns)),$   
 $j2 := (\lfloor ns ! idx \rfloor_N, 1)]$

**lemma**  $tm3$  [*transforms-intros*]:  
**assumes**  $ttt = 11 + 2 * nlength\ dummy + 2 * (nlength\ (ns ! idx))$   
**shows** *transforms*  $tm3\ tps0\ ttt\ tps3$   
*<proof>*

**definition**  $tps4 \equiv tps0$   
 $[j1 := nltape' ns\ (Suc\ idx),$   
 $j2 := (\lfloor ns ! idx \rfloor_N, 1)]$

**lemma**  $tm4$ :  
**assumes**  $ttt = 12 + 2 * nlength\ dummy + 2 * (nlength\ (ns ! idx))$   
**shows** *transforms*  $tm4\ tps0\ ttt\ tps4$   
*<proof>*

**end**

**end**

**lemma** *transforms-tm-nextract-4I* [*transforms-intros*]:  
**fixes**  $j1\ j2$  :: *tapeidx*  
**fixes**  $tps\ tps'$  :: *tape list* **and**  $k$   $idx$   $dummy$  :: *nat* **and**  $ns$  :: *nat list*  
**assumes**  $j1 < k$   $j2 < k$   $0 < j1$   $0 < j2$   $j1 \neq j2$   $length\ tps = k$   
**and**  $idx < length\ ns$   
**assumes**  
 $tps ! j1 = nltape' ns\ idx$   
 $tps ! j2 = (\lfloor dummy \rfloor_N, 1)$   
**assumes**  $ttt = 12 + 2 * nlength\ dummy + 2 * (nlength\ (ns ! idx))$   
**assumes**  $tps' = tps$   
 $[j1 := nltape' ns\ (Suc\ idx),$   
 $j2 := (\lfloor ns ! idx \rfloor_N, 1)]$   
**shows** *transforms*  $(tm-nextract\ | j1\ j2)\ tps\ ttt\ tps'$   
*<proof>*

### 2.8.3 Appending an element

The next Turing machine appends the number on tape  $j_2$  to the list of numbers on tape  $j_1$ .

**definition**  $tm\text{-}append :: tapeidx \Rightarrow tapeidx \Rightarrow machine$  **where**

```

tm-append j1 j2  $\equiv$ 
  tm-right-until j1 { $\square$ } ;;
  tm-cp-until j2 j1 { $\square$ } ;;
  tm-cr j2 ;;
  tm-char j1 |

```

**lemma**  $tm\text{-}append\text{-}tm$ :

```

assumes  $0 < j1$  and  $G \geq 5$  and  $j1 < k$  and  $j2 < k$ 
shows  $turing\text{-}machine\ k\ G\ (tm\text{-}append\ j1\ j2)$ 
<proof>

```

**locale**  $turing\text{-}machine\text{-}append =$

```

fixes  $j1\ j2 :: tapeidx$ 

```

**begin**

**definition**  $tm1 \equiv tm\text{-}right\text{-}until\ j1\ \{\square\}$

**definition**  $tm2 \equiv tm1\ ;;\ tm\text{-}cp\text{-}until\ j2\ j1\ \{\square\}$

**definition**  $tm3 \equiv tm2\ ;;\ tm\text{-}cr\ j2$

**definition**  $tm4 \equiv tm3\ ;;\ tm\text{-}char\ j1\ |$

**lemma**  $tm4\text{-}eq\text{-}tm\text{-}append$ :  $tm4 = tm\text{-}append\ j1\ j2$

```

<proof>

```

**context**

```

fixes  $tps0 :: tape\ list$  and  $k\ i1\ n :: nat$  and  $ns :: nat\ list$ 
assumes  $jk$ :  $length\ tps0 = k$   $j1 < k$   $j2 < k$   $j1 \neq j2$   $0 < j1$ 
and  $i1$ :  $i1 \leq Suc\ (nlength\ ns)$ 
and  $tps0$ :
   $tps0\ !\ j1 = (\lfloor ns \rfloor_{NL}, i1)$ 
   $tps0\ !\ j2 = (\lfloor n \rfloor_N, 1)$ 

```

**begin**

**lemma**  $k$ :  $k \geq 2$

```

<proof>

```

**lemma**  $tm1$  [ $transforms\text{-}intros$ ]:

```

assumes  $t1t = Suc\ (Suc\ (nlength\ ns) - i1)$ 
and  $tps' = tps0[j1 := nltape\ ns]$ 
shows  $transforms\ tm1\ tps0\ t1t\ tps'$ 
<proof>

```

**lemma**  $tm2$  [ $transforms\text{-}intros$ ]:

```

assumes  $t1t = 3 + nlength\ ns - i1 + nlength\ n$ 
and  $tps' = tps0$ 
   $[j1 := (\lfloor numlist\ ns\ @\ canrepr\ n \rfloor, Suc\ (nlength\ ns) + nlength\ n),$ 
   $j2 := (\lfloor n \rfloor_N, Suc\ (nlength\ n))]$ 
shows  $transforms\ tm2\ tps0\ t1t\ tps'$ 
<proof>

```

**definition**  $tps3 \equiv tps0$

```

 $[j1 := (\lfloor numlist\ ns\ @\ canrepr\ n \rfloor, Suc\ (nlength\ ns) + nlength\ n)]$ 

```

**lemma**  $tm3$  [ $transforms\text{-}intros$ ]:

```

assumes  $t1t = 6 + nlength\ ns - i1 + 2 * nlength\ n$ 
shows  $transforms\ tm3\ tps0\ t1t\ tps3$ 
<proof>

```

**definition**  $tps4 = tps0$

```

 $[j1 := (\lfloor numlist\ (ns\ @\ [n]) \rfloor, Suc\ (nlength\ (ns\ @\ [n])))]$ 

```

```

lemma tm4:
  assumes  $t1t = 7 + \text{nlength } ns - i1 + 2 * \text{nlength } n$ 
  shows transforms tm4 tps0 t1t tps4
  <proof>

```

**end**

**end**

```

lemma tm-append [transforms-intros]:
  fixes  $j1\ j2 :: \text{tapeidx}$ 
  fixes  $tps :: \text{tape list}$  and  $k\ i1\ n :: \text{nat}$  and  $ns :: \text{nat list}$ 
  assumes  $0 < j1$ 
  assumes  $\text{length } tps = k\ j1 < k\ j2 < k\ j1 \neq j2\ i1 \leq \text{Suc } (\text{nlength } ns)$ 
  and  $tps ! j1 = (\lfloor ns \rfloor_{NL}, i1)$ 
  and  $tps ! j2 = (\lfloor n \rfloor_N, 1)$ 
  assumes  $t1t = 7 + \text{nlength } ns - i1 + 2 * \text{nlength } n$ 
  and  $tps' = tps$ 
   $[j1 := \text{nltape } (ns @ [n])]$ 
  shows transforms (tm-append j1 j2) tps t1t tps'
  <proof>

```

## 2.8.4 Computing the length

The next Turing machine counts the number of terminator symbols  $z$  on tape  $j_1$  and stores the result on tape  $j_2$ . Thus, if  $j_1$  contains a list of numbers, tape  $j_2$  will contain the length of the list.

```

definition tm-count :: tapeidx  $\Rightarrow$  tapeidx  $\Rightarrow$  symbol  $\Rightarrow$  machine where
  tm-count  $j1\ j2\ z \equiv$ 
    WHILE tm-right-until  $j1\ \{\square, z\} ; \lambda rs. rs ! j1 \neq \square\ DO$ 
      tm-incr  $j2$  ;;
      tm-right  $j1$ 
    DONE ;;
    tm-cr  $j1$ 

```

```

lemma tm-count-tm:
  assumes  $0 < j2$  and  $j1 < k$  and  $j2 < k$  and  $j1 \neq j2$  and  $G \geq 4$ 
  shows turing-machine  $k\ G\ (\text{tm-count } j1\ j2\ z)$ 
  <proof>

```

```

locale turing-machine-count =
  fixes  $j1\ j2 :: \text{tapeidx}$ 
begin

```

```

definition tmC  $\equiv \text{tm-right-until } j1\ \{\square, \}$ 
definition tmB1  $\equiv \text{tm-incr } j2$ 
definition tmB2  $\equiv \text{tmB1} ; \text{tm-right } j1$ 
definition tmL  $\equiv \text{WHILE } \text{tmC} ; \lambda rs. rs ! j1 \neq \square\ DO\ \text{tmB2}\ \text{DONE}$ 
definition tm2  $\equiv \text{tmL} ; \text{tm-cr } j1$ 

```

```

lemma tm2-eq-tm-count:  $\text{tm2} = \text{tm-count } j1\ j2$  |
  <proof>

```

```

context
  fixes  $tps0 :: \text{tape list}$  and  $k :: \text{nat}$  and  $ns :: \text{nat list}$ 
  assumes  $jk: j1 < k\ j2 < k\ 0 < j2\ j1 \neq j2\ \text{length } tps0 = k$ 
  and tps0:
     $tps0 ! j1 = (\lfloor ns \rfloor_{NL}, 1)$ 
     $tps0 ! j2 = (\lfloor 0 \rfloor_N, 1)$ 
begin

```

```

definition tpsL  $t \equiv tps0$ 
   $[j1 := (\lfloor ns \rfloor_{NL}, \text{Suc } (\text{nlength } (\text{take } t\ ns)))]$ ,

```

$j2 := (\lfloor t \rfloor_N, 1)$

**definition**  $tpsC\ t \equiv tps0$

$[j1 := (\lfloor ns \rfloor_{NL}, \text{if } t < \text{length } ns \text{ then } nlength (\text{take } (Suc\ t)\ ns) \text{ else } Suc\ (nlength\ ns)),$   
 $j2 := (\lfloor t \rfloor_N, 1)]$

**lemma**  $tmC$ :

**assumes**  $t \leq \text{length } ns$

**and**  $t = Suc\ (\text{if } t = \text{length } ns \text{ then } 0 \text{ else } nlength\ (ns\ !\ t))$

**shows**  $\text{transforms } tmC\ (tpsL\ t)\ ttt\ (tpsC\ t)$

$\langle \text{proof} \rangle$

**lemma**  $tmC'$  [*transforms-intros*]:

**assumes**  $t \leq \text{length } ns$

**shows**  $\text{transforms } tmC\ (tpsL\ t)\ (Suc\ (nlength\ ns))\ (tpsC\ t)$

$\langle \text{proof} \rangle$

**definition**  $tpsB1\ t \equiv tps0$

$[j1 := (\lfloor ns \rfloor_{NL}, nlength\ (\text{take } (Suc\ t)\ ns)),$

$j2 := (\lfloor Suc\ t \rfloor_N, 1)]$

**lemma**  $tmB1$  [*transforms-intros*]:

**assumes**  $t < \text{length } ns$  **and**  $t = 5 + 2 * nlength\ t$

**shows**  $\text{transforms } tmB1\ (tpsC\ t)\ ttt\ (tpsB1\ t)$

$\langle \text{proof} \rangle$

**definition**  $tpsB2\ t \equiv tps0$

$[j1 := (\lfloor ns \rfloor_{NL}, Suc\ (nlength\ (\text{take } (Suc\ t)\ ns))),$

$j2 := (\lfloor Suc\ t \rfloor_N, 1)]$

**lemma**  $tmB2$ :

**assumes**  $t < \text{length } ns$  **and**  $t = 6 + 2 * nlength\ t$

**shows**  $\text{transforms } tmB2\ (tpsC\ t)\ ttt\ (tpsB2\ t)$

$\langle \text{proof} \rangle$

**lemma**  $tpsB2\text{-eq-tpsL}$ :  $tpsB2\ t = tpsL\ (Suc\ t)$

$\langle \text{proof} \rangle$

**lemma**  $tmB2'$  [*transforms-intros*]:

**assumes**  $t < \text{length } ns$

**shows**  $\text{transforms } tmB2\ (tpsC\ t)\ (6 + 2 * nlength\ ns)\ (tpsL\ (Suc\ t))$

$\langle \text{proof} \rangle$

**lemma**  $tmL$  [*transforms-intros*]:

**assumes**  $t = 13 * nlength\ ns \wedge 2 + 2$

**shows**  $\text{transforms } tmL\ (tpsL\ 0)\ ttt\ (tpsC\ (\text{length } ns))$

$\langle \text{proof} \rangle$

**definition**  $tps2 \equiv tps0$

$[j2 := (\lfloor \text{length } ns \rfloor_N, 1)]$

**lemma**  $tm2$ :

**assumes**  $t = 13 * nlength\ ns \wedge 2 + 5 + nlength\ ns$

**shows**  $\text{transforms } tm2\ (tpsL\ 0)\ ttt\ tps2$

$\langle \text{proof} \rangle$

**lemma**  $tpsL\text{-eq-tps0}$ :  $tpsL\ 0 = tps0$

$\langle \text{proof} \rangle$

**lemma**  $tm2'$ :

**assumes**  $t = 14 * nlength\ ns \wedge 2 + 5$

**shows**  $\text{transforms } tm2\ tps0\ ttt\ tps2$

$\langle \text{proof} \rangle$

end

end

**lemma** *transforms-tm-count-4I* [*transforms-intros*]:  
 **fixes**  $j1\ j2 :: \text{tapeidx}$   
 **fixes**  $tps\ tps' :: \text{tape list}$  **and**  $k :: \text{nat}$  **and**  $ns :: \text{nat list}$   
 **assumes**  $j1 < k\ j2 < k\ 0 < j2\ j1 \neq j2\ \text{length } tps = k$   
 **assumes**  
  $tps ! j1 = (\lfloor ns \rfloor_{NL}, 1)$   
  $tps ! j2 = (\lfloor 0 \rfloor_N, 1)$   
 **assumes**  $ttt = 14 * \text{nlength } ns \wedge 2 + 5$   
 **assumes**  $tps' = tps[j2 := (\lfloor \text{length } ns \rfloor_N, 1)]$   
 **shows** *transforms* (*tm-count*  $j1\ j2\ |$ )  $tps\ ttt\ tps'$   
(*proof*)

## 2.8.5 Extracting the $n$ -th element

The next Turing machine expects a list on tape  $j_1$  and an index  $i$  on  $j_2$  and writes the  $i$ -th element of the list to  $j_2$ , overwriting  $i$ . The TM does not terminate for out-of-bounds access, which of course we will never attempt. Again the parameter  $z$  is a generic terminator symbol.

**definition** *tm-nth-inplace*  $:: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{symbol} \Rightarrow \text{machine}$  **where**  
 *tm-nth-inplace*  $j1\ j2\ z \equiv$   
 *WHILE*  $\square ; \lambda rs. rs ! j2 \neq \square \ \square \ \text{DO}$   
 *tm-decr*  $j2$  ;;  
 *tm-right-until*  $j1\ \{z\}$  ;;  
 *tm-right*  $j1$   
 *DONE* ;;  
 *tm-cp-until*  $j1\ j2\ \{z\}$  ;;  
 *tm-cr*  $j2$  ;;  
 *tm-cr*  $j1$

**lemma** *tm-nth-inplace-tm*:  
 **assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$   
 **shows** *turing-machine*  $k\ G\ (\text{tm-nth-inplace } j1\ j2\ |)$   
(*proof*)

**locale** *turing-machine-nth-inplace* =  
 **fixes**  $j1\ j2 :: \text{tapeidx}$   
**begin**

**definition**  $tmL1 \equiv \text{tm-decr } j2$   
**definition**  $tmL2 \equiv tmL1$  ;; *tm-right-until*  $j1\ \{\}$   
**definition**  $tmL3 \equiv tmL2$  ;; *tm-right*  $j1$   
**definition**  $tmL \equiv \text{WHILE } \square ; \lambda rs. rs ! j2 \neq \square \ \square \ \text{DO } tmL3 \ \text{DONE}$   
**definition**  $tm2 \equiv tmL$  ;; *tm-cp-until*  $j1\ j2\ \{\}$   
**definition**  $tm3 \equiv tm2$  ;; *tm-cr*  $j2$   
**definition**  $tm4 \equiv tm3$  ;; *tm-cr*  $j1$

**lemma** *tm4-eq-tm-nth-inplace*:  $tm4 = \text{tm-nth-inplace } j1\ j2\ |$   
(*proof*)

**context**  
 **fixes**  $tps0 :: \text{tape list}$  **and**  $k\ idx :: \text{nat}$  **and**  $ns :: \text{nat list}$   
 **assumes**  $jk: j1 < k\ j2 < k\ 0 < j2\ j1 \neq j2\ \text{length } tps0 = k$   
 **and**  $idx: idx < \text{length } ns$   
 **and**  $tps0$ :  
  $tps0 ! j1 = (\lfloor ns \rfloor_{NL}, 1)$   
  $tps0 ! j2 = (\lfloor idx \rfloor_N, 1)$   
**begin**

**definition**  $tpsL\ t \equiv tps0$

$[j1 := (\lfloor ns \rfloor_{NL}, \text{Suc} (\text{nlength} (\text{take } t \text{ ns}))),$   
 $j2 := (\lfloor idx - t \rfloor_N, 1)]$

**definition**  $\text{tpsL1 } t \equiv \text{tps0}$

$[j1 := (\lfloor ns \rfloor_{NL}, \text{Suc} (\text{nlength} (\text{take } t \text{ ns}))),$   
 $j2 := (\lfloor idx - t - 1 \rfloor_N, 1)]$

**lemma**  $\text{tmL1}$  [*transforms-intros*]:

**assumes**  $\text{t} = 8 + 2 * \text{nlength} (\text{idx} - t)$   
**shows**  $\text{transforms } \text{tmL1} (\text{tpsL } t) \text{ ttt} (\text{tpsL1 } t)$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tpsL2 } t \equiv \text{tps0}$

$[j1 := (\lfloor ns \rfloor_{NL}, \text{nlength} (\text{take} (\text{Suc } t) \text{ ns})),$   
 $j2 := (\lfloor idx - t - 1 \rfloor_N, 1)]$

**lemma**  $\text{tmL2}$ :

**assumes**  $t < \text{length } ns$  **and**  $\text{t} = 8 + 2 * \text{nlength} (\text{idx} - t) + \text{Suc} (\text{nlength} (ns ! t))$   
**shows**  $\text{transforms } \text{tmL2} (\text{tpsL } t) \text{ ttt} (\text{tpsL2 } t)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{tmL2}'$  [*transforms-intros*]:

**assumes**  $t < \text{length } ns$  **and**  $\text{t} = 9 + 2 * \text{nlength } idx + \text{nlength} (\text{Max} (\text{set } ns))$   
**shows**  $\text{transforms } \text{tmL2} (\text{tpsL } t) \text{ ttt} (\text{tpsL2 } t)$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tpsL3 } t \equiv \text{tps0}$

$[j1 := (\lfloor ns \rfloor_{NL}, \text{Suc} (\text{nlength} (\text{take} (\text{Suc } t) \text{ ns}))),$   
 $j2 := (\lfloor idx - t - 1 \rfloor_N, 1)]$

**lemma**  $\text{tmL3}$ :

**assumes**  $t < \text{length } ns$  **and**  $\text{t} = 10 + 2 * \text{nlength } idx + \text{nlength} (\text{Max} (\text{set } ns))$   
**shows**  $\text{transforms } \text{tmL3} (\text{tpsL } t) \text{ ttt} (\text{tpsL3 } t)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{tpsL3-eq-tpsL}$ :  $\text{tpsL3 } t = \text{tpsL} (\text{Suc } t)$

$\langle \text{proof} \rangle$

**lemma**  $\text{tmL}$ :

**assumes**  $\text{t} = idx * (12 + 2 * \text{nlength } idx + \text{nlength} (\text{Max} (\text{set } ns))) + 1$   
**shows**  $\text{transforms } \text{tmL} (\text{tpsL } 0) \text{ ttt} (\text{tpsL } idx)$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tps1} \equiv \text{tps0}$

$[j1 := (\lfloor ns \rfloor_{NL}, \text{Suc} (\text{nlength} (\text{take } idx \text{ ns}))),$   
 $j2 := (\lfloor 0 \rfloor_N, 1)]$

**lemma**  $\text{tps1-eq-tpsL}$ :  $\text{tps1} = \text{tpsL } idx$

$\langle \text{proof} \rangle$

**lemma**  $\text{tps0-eq-tpsL}$ :  $\text{tps0} = \text{tpsL } 0$

$\langle \text{proof} \rangle$

**lemma**  $\text{tmL}'$  [*transforms-intros*]:

**assumes**  $\text{t} = idx * (12 + 2 * \text{nlength } idx + \text{nlength} (\text{Max} (\text{set } ns))) + 1$   
**shows**  $\text{transforms } \text{tmL} \text{ tps0} \text{ ttt} \text{ tps1}$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tps2} \equiv \text{tps0}$

$[j1 := (\lfloor ns \rfloor_{NL}, \text{nlength} (\text{take} (\text{Suc } idx) \text{ ns})),$   
 $j2 := (\lfloor ns ! idx \rfloor_N, \text{Suc} (\text{nlength} (ns ! idx)))]$

**lemma**  $\text{tm2}$  [*transforms-intros*]:

**assumes**  $ttt = idx * (12 + 2 * nlength\ idx + nlength\ (Max\ (set\ ns))) + 2 + nlength\ (ns\ !\ idx)$   
**shows**  $transforms\ tm2\ tps0\ ttt\ tps2$   
 $\langle proof \rangle$

**definition**  $tps3 \equiv tps0$   
 $[j1 := (\lfloor ns \rfloor_{NL}, nlength\ (take\ (Suc\ idx)\ ns)),$   
 $j2 := (\lfloor ns\ !\ idx \rfloor_N, 1)]$

**lemma**  $tm3$  [*transforms-intros*]:  
**assumes**  $ttt = idx * (12 + 2 * nlength\ idx + nlength\ (Max\ (set\ ns))) + 5 + 2 * nlength\ (ns\ !\ idx)$   
**shows**  $transforms\ tm3\ tps0\ ttt\ tps3$   
 $\langle proof \rangle$

**definition**  $tps4 \equiv tps0$   
 $[j2 := (\lfloor ns\ !\ idx \rfloor_N, 1)]$

**lemma**  $tm4$ :  
**assumes**  $ttt = idx * (12 + 2 * nlength\ idx + nlength\ (Max\ (set\ ns))) + 7 + 2 * nlength\ (ns\ !\ idx) + nlength\ (take\ (Suc\ idx)\ ns)$   
**shows**  $transforms\ tm4\ tps0\ ttt\ tps4$   
 $\langle proof \rangle$

**lemma**  $tm4'$ :  
**assumes**  $ttt = 18 * nlength\ ns^2 + 12$   
**shows**  $transforms\ tm4\ tps0\ ttt\ tps4$   
 $\langle proof \rangle$

**end**

**end**

**lemma**  $transforms-tm-nth-inplace-4I$  [*transforms-intros*]:  
**fixes**  $j1\ j2 :: tapeidx$   
**fixes**  $tps\ tps' :: tape\ list$  **and**  $k\ idx :: nat$  **and**  $ns :: nat\ list$   
**assumes**  $j1 < k\ j2 < k\ 0 < j2\ j1 \neq j2\ length\ tps = k$   
**and**  $idx < length\ ns$   
**assumes**  
 $tps\ !\ j1 = (\lfloor ns \rfloor_{NL}, 1)$   
 $tps\ !\ j2 = (\lfloor idx \rfloor_N, 1)$   
**assumes**  $ttt = 18 * nlength\ ns^2 + 12$   
**assumes**  $tps' = tps$   
 $[j2 := (\lfloor ns\ !\ idx \rfloor_N, 1)]$   
**shows**  $transforms\ (tm-nth-inplace\ j1\ j2\ |)\ tps\ ttt\ tps'$   
 $\langle proof \rangle$

The next Turing machine expects a list on tape  $j_1$  and an index  $i$  on tape  $j_2$ . It writes the  $i$ -th element of the list to tape  $j_3$ . Like the previous TM, it will not terminate on out-of-bounds access, and  $z$  is a parameter for the symbol that terminates the list elements.

**definition**  $tm-nth :: tapeidx \Rightarrow tapeidx \Rightarrow tapeidx \Rightarrow symbol \Rightarrow machine$  **where**  
 $tm-nth\ j1\ j2\ j3\ z \equiv$   
 $tm-copy\ j2\ j3\ ;;$   
 $tm-nth-inplace\ j1\ j3\ z$

**lemma**  $tm-nth-tm$ :  
**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j2\ 0 < j1\ j1 < k\ j2 < k\ 0 < j3\ j3 < k\ j2 \neq j3$   
**shows**  $turing-machine\ k\ G\ (tm-nth\ j1\ j2\ j3\ |)$   
 $\langle proof \rangle$

**lemma**  $transforms-tm-nth-4I$  [*transforms-intros*]:  
**fixes**  $j1\ j2\ j3 :: tapeidx$   
**fixes**  $tps\ tps' :: tape\ list$  **and**  $k\ idx :: nat$  **and**  $ns :: nat\ list$   
**assumes**  $j1 < k\ j2 < k\ j3 < k\ 0 < j1\ 0 < j2\ 0 < j3\ j1 \neq j2\ j2 \neq j3\ j1 \neq j3$   
**and**  $length\ tps = k$

```

and  $idx < \text{length } ns$ 
assumes
   $tps ! j1 = (\lfloor ns \rfloor_{NL}, 1)$ 
   $tps ! j2 = (\lfloor idx \rfloor_N, 1)$ 
   $tps ! j3 = (\lfloor 0 \rfloor_N, 1)$ 
assumes  $t1t = 21 * \text{nlength } ns \wedge 2 + 26$ 
assumes  $tps' = tps$ 
   $[j3 := (\lfloor ns ! idx \rfloor_N, 1)]$ 
shows  $\text{transforms } (tm\text{-nth } j1 \ j2 \ j3 \ |) \ tps \ t1t \ tps'$ 
<proof>

```

## 2.8.6 Finding the previous position of an element

The Turing machine in this section implements a slightly peculiar functionality, which we will start using only in Section 6. Given a list of numbers and an index  $i$  it determines the greatest index less than  $i$  where the list contains the same element as in position  $i$ . If no such element exists, it returns  $i$ . Formally:

```

definition  $\text{previous} :: \text{nat list} \Rightarrow \text{nat} \Rightarrow \text{nat}$  where
   $\text{previous } ns \ idx \equiv$ 
     $\text{if } \exists i < idx. ns ! i = ns ! idx$ 
     $\text{then } \text{GREATEST } i. i < idx \wedge ns ! i = ns ! idx$ 
     $\text{else } idx$ 

```

```

lemma  $\text{previous-less}$ :
assumes  $\exists i < idx. ns ! i = ns ! idx$ 
shows  $\text{previous } ns \ idx < idx \wedge ns ! (\text{previous } ns \ idx) = ns ! idx$ 
<proof>

```

```

lemma  $\text{previous-eq}$ :  $\text{previous } ns \ idx = idx \iff \neg (\exists i < idx. ns ! i = ns ! idx)$ 
<proof>

```

```

lemma  $\text{previous-le}$ :  $\text{previous } ns \ idx \leq idx$ 
<proof>

```

The following Turing machine expects the list on tape  $j_1$  and the index on tape  $j_2$ . It outputs the result on tape  $j_2 + 5$ . The tapes  $j_2 + 1, \dots, j_2 + 4$  are scratch space.

```

definition  $\text{tm-prev} :: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  where
   $\text{tm-prev } j1 \ j2 \equiv$ 
     $\text{tm-copyn } j2 \ (j2 + 5) \ ;;$ 
     $\text{tm-nth } j1 \ j2 \ (j2 + 1) \ | \ ;;$ 
     $\text{WHILE } \text{tm-equalsn } (j2 + 2) \ j2 \ (j2 + 4) \ ; \ \lambda rs. rs ! (j2 + 4) = \square \ \text{DO}$ 
     $\text{tm-nth } j1 \ (j2 + 2) \ (j2 + 3) \ | \ ;;$ 
     $\text{tm-equalsn } (j2 + 1) \ (j2 + 3) \ (j2 + 4) \ ;;$ 
     $\text{tm-setn } (j2 + 3) \ 0 \ ;;$ 
     $\text{IF } \lambda rs. rs ! (j2 + 4) \neq \square \ \text{THEN}$ 
     $\text{tm-setn } (j2 + 4) \ 0 \ ;;$ 
     $\text{tm-copyn } (j2 + 2) \ (j2 + 5)$ 
     $\text{ELSE}$ 
     $\square$ 
     $\text{ENDIF} \ ;;$ 
     $\text{tm-incr } (j2 + 2)$ 
     $\text{DONE} \ ;;$ 
     $\text{tm-erase-cr } (j2 + 1) \ ;;$ 
     $\text{tm-erase-cr } (j2 + 2) \ ;;$ 
     $\text{tm-erase-cr } (j2 + 4)$ 

```

```

lemma  $\text{tm-prev-tm}$ :
assumes  $k \geq j2 + 6$  and  $G \geq 4$  and  $j1 < j2$  and  $0 < j1$ 
shows  $\text{turing-machine } k \ G \ (\text{tm-prev } j1 \ j2)$ 
<proof>

```

```

locale  $\text{turing-machine-prev} =$ 
fixes  $j1 \ j2 :: \text{tapeidx}$ 

```

**begin**

**definition**  $tm1 \equiv tm\text{-copyn } j2 \ (j2 + 5)$   
**definition**  $tm2 \equiv tm1 \ ; \ ; \ tm\text{-nth } j1 \ j2 \ (j2 + 1) \ |$   
**definition**  $tmC \equiv tm\text{-equalsn } (j2 + 2) \ j2 \ (j2 + 4)$   
**definition**  $tmB1 \equiv tm\text{-nth } j1 \ (j2 + 2) \ (j2 + 3) \ |$   
**definition**  $tmB2 \equiv tmB1 \ ; \ ; \ tm\text{-equalsn } (j2 + 1) \ (j2 + 3) \ (j2 + 4)$   
**definition**  $tmB3 \equiv tmB2 \ ; \ ; \ tm\text{-setn } (j2 + 3) \ 0$   
**definition**  $tmI1 \equiv tm\text{-setn } (j2 + 4) \ 0$   
**definition**  $tmI2 \equiv tmI1 \ ; \ ; \ tm\text{-copyn } (j2 + 2) \ (j2 + 5)$   
**definition**  $tmI \equiv IF \ \lambda rs. \ rs \ ! \ (j2 + 4) \neq \square \ THEN \ tmI2 \ ELSE \ [] \ ENDIF$   
**definition**  $tmB4 \equiv tmB3 \ ; \ ; \ tmI$   
**definition**  $tmB5 \equiv tmB4 \ ; \ ; \ tm\text{-incr } (j2 + 2)$   
**definition**  $tmL \equiv WHILE \ tmC \ ; \ \lambda rs. \ rs \ ! \ (j2 + 4) = \square \ DO \ tmB5 \ DONE$   
**definition**  $tm3 \equiv tm2 \ ; \ ; \ tmL$   
**definition**  $tm4 \equiv tm3 \ ; \ ; \ tm\text{-erase-cr } (j2 + 1)$   
**definition**  $tm5 \equiv tm4 \ ; \ ; \ tm\text{-erase-cr } (j2 + 2)$   
**definition**  $tm6 \equiv tm5 \ ; \ ; \ tm\text{-erase-cr } (j2 + 4)$

**lemma**  $tm6\text{-eq-}tm\text{-prev}$ :  $tm6 = tm\text{-prev } j1 \ j2$   
*<proof>*

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $k \ idx :: \text{nat}$  **and**  $ns :: \text{nat list}$   
**assumes**  $jk$ :  $0 < j1 \ j1 < j2 \ j2 + 6 \leq k \ \text{length } tps0 = k$   
**and**  $idx$ :  $idx < \text{length } ns$   
**and**  $tps0$ :  
 $tps0 \ ! \ j1 = ([ns]_{NL}, 1)$   
 $tps0 \ ! \ j2 = ([idx]_N, 1)$   
 $tps0 \ ! \ (j2 + 1) = ([0]_N, 1)$   
 $tps0 \ ! \ (j2 + 2) = ([0]_N, 1)$   
 $tps0 \ ! \ (j2 + 3) = ([0]_N, 1)$   
 $tps0 \ ! \ (j2 + 4) = ([0]_N, 1)$   
 $tps0 \ ! \ (j2 + 5) = ([0]_N, 1)$

**begin**

**definition**  $tps1 \equiv tps0$   
 $[j2 + 5 := ([idx]_N, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:  
**assumes**  $ttt = 14 + 3 * \text{nlength } idx$   
**shows**  $\text{transforms } tm1 \ tps0 \ ttt \ tps1$   
*<proof>*

**definition**  $tps2 \equiv tps0$   
 $[j2 + 1 := ([ns \ ! \ idx]_N, 1),$   
 $j2 + 5 := ([idx]_N, 1)]$

**lemma**  $tm2$ :  
**assumes**  $ttt = 14 + 3 * \text{nlength } idx + (21 * (\text{nlength } ns)^2 + 26)$   
**shows**  $\text{transforms } tm2 \ tps0 \ ttt \ tps2$   
*<proof>*

**definition**  $rv :: \text{nat} \Rightarrow \text{nat}$  **where**  
 $rv \ t \equiv \text{if } \exists i < t. \ ns \ ! \ i = ns \ ! \ idx \ \text{then } \text{GREATEST } i. \ i < t \wedge \ ns \ ! \ i = ns \ ! \ idx \ \text{else } idx$

**lemma**  $rv\text{-}0$ :  $rv \ 0 = idx$   
*<proof>*

**lemma**  $rv\text{-}le$ :  $rv \ t \leq \max \ t \ idx$   
*<proof>*

**lemma**  $rv\text{-}change$ :

**assumes**  $t < \text{length } ns$  **and**  $idx < \text{length } ns$  **and**  $ns ! t = ns ! idx$   
**shows**  $rv (Suc t) = t$   
 $\langle \text{proof} \rangle$

**lemma** *rv-keep*:  
**assumes**  $t < \text{length } ns$  **and**  $idx < \text{length } ns$  **and**  $ns ! t \neq ns ! idx$   
**shows**  $rv (Suc t) = rv t$   
 $\langle \text{proof} \rangle$

**definition** *tpsL* ::  $nat \Rightarrow \text{tape list}$  **where**  
 $tpsL\ t \equiv tps0$   
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 5 := ([rv\ t]_N, 1)]$

**lemma** *tpsL-eq-tps2*:  $tpsL\ 0 = tps2$   
 $\langle \text{proof} \rangle$

**definition** *tpsC* ::  $nat \Rightarrow \text{tape list}$  **where**  
 $tpsC\ t \equiv tps0$   
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 4 := ([t = idx]_B, 1),$   
 $j2 + 5 := ([rv\ t]_N, 1)]$

**lemma** *tmC*:  
**assumes**  $ttt = 3 * nlength (min\ t\ idx) + 7$   
**shows** *transforms* *tmC* (*tpsL*  $t$ )  $ttt$  (*tpsC*  $t$ )  
 $\langle \text{proof} \rangle$

**lemma** *tmC'* [*transforms-intros*]:  
**assumes**  $ttt = 3 * nlength\ ns \wedge 2 + 7$  **and**  $t \leq idx$   
**shows** *transforms* *tmC* (*tpsL*  $t$ )  $ttt$  (*tpsC*  $t$ )  
 $\langle \text{proof} \rangle$

**lemma** *condition-tpsC*:  $(read (tpsC\ t)) ! (j2 + 4) \neq \square \longleftrightarrow t = idx$   
 $\langle \text{proof} \rangle$

**definition** *tpsB1*  $t \equiv tps0$   
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 3 := ([ns ! t]_N, 1),$   
 $j2 + 4 := ([t = idx]_B, 1),$   
 $j2 + 5 := ([rv\ t]_N, 1)]$

**lemma** *tmB1* [*transforms-intros*]:  
**assumes**  $ttt = 21 * (nlength\ ns)^2 + 26$  **and**  $t < idx$   
**shows** *transforms* *tmB1* (*tpsC*  $t$ )  $ttt$  (*tpsB1*  $t$ )  
 $\langle \text{proof} \rangle$

**definition** *tpsB2*  $t \equiv tps0$   
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 3 := ([ns ! t]_N, 1),$   
 $j2 + 4 := ([ns ! idx = ns ! t]_B, 1),$   
 $j2 + 5 := ([rv\ t]_N, 1)]$

**lemma** *tmB2*:  
**assumes**  $ttt = 21 * (nlength\ ns)^2 + 26 + (3 * nlength (min (ns ! idx) (ns ! t)) + 7)$   
**and**  $t < idx$   
**shows** *transforms* *tmB2* (*tpsC*  $t$ )  $ttt$  (*tpsB2*  $t$ )  
 $\langle \text{proof} \rangle$

**lemma** *tmB2'* [*transforms-intros*]:  
**assumes**  $t \equiv 24 * (\text{nlength } ns)^2 + 33$  **and**  $t < idx$   
**shows** *transforms tmB2 (tpsC t) ttt (tpsB2 t)*  
*<proof>*

**definition** *tpsB3 t*  $\equiv$  *tps0*  
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 4 := ([ns ! idx = ns ! t]_B, 1),$   
 $j2 + 5 := ([rv t]_N, 1)]$

**lemma** *condition-tpsB3*:  $(\text{read } (tpsB3 t)) ! (j2 + 4) \neq \square \iff ns ! idx = ns ! t$   
*<proof>*

**lemma** *tmB3* [*transforms-intros*]:  
**assumes**  $t \equiv 24 * (\text{nlength } ns)^2 + 33 + (10 + 2 * \text{nlength } (ns ! t))$  **and**  $t < idx$   
**shows** *transforms tmB3 (tpsC t) ttt (tpsB3 t)*  
*<proof>*

**definition** *tpsI0 t*  $\equiv$  *tps0*  
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 4 := ([I]_N, 1),$   
 $j2 + 5 := ([rv t]_N, 1)]$

**definition** *tpsI1 t*  $\equiv$  *tps0*  
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 5 := ([rv t]_N, 1)]$

**lemma** *tmI1* [*transforms-intros*]:  
**assumes**  $t < idx$  **and**  $ns ! idx = ns ! t$   
**shows** *transforms tmI1 (tpsB3 t) I2 (tpsI1 t)*  
*<proof>*

**definition** *tpsI2 t*  $\equiv$  *tps0*  
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 5 := ([t]_N, 1)]$

**lemma** *tmI2* [*transforms-intros*]:  
**assumes**  $t \equiv 26 + 3 * \text{nlength } t + 3 * \text{nlength } (rv t)$   
**and**  $t < idx$   
**and**  $ns ! idx = ns ! t$   
**shows** *transforms tmI2 (tpsB3 t) ttt (tpsI2 t)*  
*<proof>*

**definition** *tpsI t*  $\equiv$  *tps0*  
 $[j2 + 1 := ([ns ! idx]_N, 1),$   
 $j2 + 2 := ([t]_N, 1),$   
 $j2 + 5 := ([rv (Suc t)]_N, 1)]$

**lemma** *tmI* [*transforms-intros*]:  
**assumes**  $t \equiv 28 + 6 * \text{nlength } ns$  **and**  $t < idx$   
**shows** *transforms tmI (tpsB3 t) ttt (tpsI t)*  
*<proof>*

**lemma** *tmB4*:  
**assumes**  $t \equiv 71 + 24 * (\text{nlength } ns)^2 + 2 * \text{nlength } (ns ! t) + 6 * \text{nlength } ns$   
**and**  $t < idx$   
**shows** *transforms tmB4 (tpsC t) ttt (tpsI t)*  
*<proof>*

**lemma** *tmB4'* [*transforms-intros*]:  
**assumes**  $ttt = 71 + 32 * (nlength\ ns)^2$  **and**  $t < idx$   
**shows** *transforms tmB4 (tpsC t) ttt (tpsI t)*  
*<proof>*

**definition** *tpsB5 t*  $\equiv$  *tps0*  
 $[j2 + 1 := ([ns\ !\ idx]_N, 1),$   
 $j2 + 2 := ([Suc\ t]_N, 1),$   
 $j2 + 5 := ([rv\ (Suc\ t)]_N, 1)]$

**lemma** *tmB5*:  
**assumes**  $ttt = 76 + 32 * (nlength\ ns)^2 + 2 * nlength\ t$  **and**  $t < idx$   
**shows** *transforms tmB5 (tpsC t) ttt (tpsB5 t)*  
*<proof>*

**lemma** *tmB5'* [*transforms-intros*]:  
**assumes**  $ttt = 76 + 34 * (nlength\ ns)^2$  **and**  $t < idx$   
**shows** *transforms tmB5 (tpsC t) ttt (tpsL (Suc t))*  
*<proof>*

**lemma** *tmL* [*transforms-intros*]:  
**assumes**  $ttt = 125 * nlength\ ns \wedge^3 + 8$  **and**  $iidx = idx$   
**shows** *transforms tmL (tpsL 0) ttt (tpsC iidx)*  
*<proof>*

**lemma** *tm2'* [*transforms-intros*]:  
**assumes**  $ttt = 14 + 3 * nlength\ idx + (21 * (nlength\ ns)^2 + 26)$   
**shows** *transforms tm2 tps0 ttt (tpsL 0)*  
*<proof>*

**lemma** *tm3* [*transforms-intros*]:  
**assumes**  $ttt = 40 + (3 * nlength\ idx + 21 * (nlength\ ns)^2) + (125 * nlength\ ns \wedge^3 + 8)$   
**shows** *transforms tm3 tps0 ttt (tpsC idx)*  
*<proof>*

**lemma** *tpsC-idx*:  
 $tpsC\ idx = tps0$   
 $[j2 + 1 := ([ns\ !\ idx]_N, 1),$   
 $j2 + 2 := ([idx]_N, 1),$   
 $j2 + 4 := ([I]_N, 1),$   
 $j2 + 5 := ([if\ \exists i < idx. ns\ !\ i = ns\ !\ idx\ then\ GREATEST\ i. i < idx \wedge ns\ !\ i = ns\ !\ idx\ else\ idx]_N, 1)]$   
*<proof>*

**definition** *tps4* :: *tape list where*  
 $tps4 \equiv tps0$   
 $[j2 + 1 := ([[]], 1),$   
 $j2 + 2 := ([idx]_N, 1),$   
 $j2 + 4 := ([I]_N, 1),$   
 $j2 + 5 := ([if\ \exists i < idx. ns\ !\ i = ns\ !\ idx\ then\ GREATEST\ i. i < idx \wedge ns\ !\ i = ns\ !\ idx\ else\ idx]_N, 1)]$

**lemma** *tm4* [*transforms-intros*]:  
**assumes**  $ttt = 55 + 3 * nlength\ idx + 21 * (nlength\ ns)^2 + 125 * nlength\ ns \wedge^3 + 2 * nlength\ (ns\ !\ idx)$   
**shows** *transforms tm4 tps0 ttt tps4*  
*<proof>*

**definition** *tps5* :: *tape list where*  
 $tps5 \equiv tps0$   
 $[j2 + 1 := ([[]], 1),$   
 $j2 + 2 := ([[]], 1),$   
 $j2 + 4 := ([I]_N, 1),$   
 $j2 + 5 := ([if\ \exists i < idx. ns\ !\ i = ns\ !\ idx\ then\ GREATEST\ i. i < idx \wedge ns\ !\ i = ns\ !\ idx\ else\ idx]_N, 1)]$

**lemma** *tm5* [*transforms-intros*]:

**assumes**  $ttt = 62 + 5 * nlength\ idx + 21 * (nlength\ ns)^2 + 125 * nlength\ ns \wedge 3 + 2 * nlength\ (ns\ !\ idx)$   
**shows** *transforms tm5 tps0 ttt tps5*  
 ⟨*proof*⟩

**definition** *tps6* :: *tape list where*

$tps6 \equiv tps0$   
 $[j2 + 1 := ([\ ], 1),$   
 $j2 + 2 := ([\ ], 1),$   
 $j2 + 4 := ([\ ], 1),$   
 $j2 + 5 := ([\ if\ \exists\ i < idx.\ ns\ !\ i = ns\ !\ idx\ then\ GREATEST\ i.\ i < idx \wedge ns\ !\ i = ns\ !\ idx\ else\ idx ]_N, 1)]$

**lemma** *tm6*:

**assumes**  $ttt = 71 + 5 * nlength\ idx + 21 * (nlength\ ns)^2 + 125 * nlength\ ns \wedge 3 + 2 * nlength\ (ns\ !\ idx)$   
**shows** *transforms tm6 tps0 ttt tps6*  
 ⟨*proof*⟩

**definition** *tps6'* :: *tape list where*

$tps6' \equiv tps0$   
 $[j2 + 5 := ([\ if\ \exists\ i < idx.\ ns\ !\ i = ns\ !\ idx\ then\ GREATEST\ i.\ i < idx \wedge ns\ !\ i = ns\ !\ idx\ else\ idx ]_N, 1)]$

**lemma** *tps6'-eq-tps6*:  $tps6' = tps6$

⟨*proof*⟩

**lemma** *tm6'*:

**assumes**  $ttt = 71 + 153 * nlength\ ns \wedge 3$   
**shows** *transforms tm6 tps0 ttt tps6'*  
 ⟨*proof*⟩

**end**

**end**

**lemma** *transforms-tm-prevI* [*transforms-intros*]:

**fixes**  $j1\ j2$  :: *tapeidx*  
**fixes**  $tps\ tps'$  :: *tape list and k idx :: nat and ns :: nat list*  
**assumes**  $0 < j1\ j1 < j2\ j2 + 6 \leq k\ length\ tps = k$   
**and**  $idx < length\ ns$   
**assumes**  
 $tps\ !\ j1 = ([ns]_{NL}, 1)$   
 $tps\ !\ j2 = ([idx]_N, 1)$   
 $tps\ !\ (j2 + 1) = ([0]_N, 1)$   
 $tps\ !\ (j2 + 2) = ([0]_N, 1)$   
 $tps\ !\ (j2 + 3) = ([0]_N, 1)$   
 $tps\ !\ (j2 + 4) = ([0]_N, 1)$   
 $tps\ !\ (j2 + 5) = ([0]_N, 1)$   
**assumes**  $ttt = 71 + 153 * nlength\ ns \wedge 3$   
**assumes**  $tps' = tps$   
 $[j2 + 5 := ([previous\ ns\ idx]_N, 1)]$   
**shows** *transforms (tm-prev j1 j2) tps ttt tps'*  
 ⟨*proof*⟩

## 2.8.7 Checking containment in a list

A Turing machine that checks whether a number given on tape  $j_2$  is contained in the list of numbers on tape  $j_1$ . If so, it writes a 1 to tape  $j_3$ , and otherwise leaves tape  $j_3$  unchanged. So tape  $j_3$  should be initialized with 0.

**definition** *tm-contains* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine where*

$tm\_contains\ j1\ j2\ j3 \equiv$   
 WHILE  $\square$  ;  $\lambda rs.\ rs\ !\ j1 \neq \square$  DO  
 $tm\_nexttract\ | j1\ (j3 + 1)$  ;;  
 $tm\_equalsn\ j2\ (j3 + 1)\ (j3 + 2)$  ;;  
 IF  $\lambda rs.\ rs\ !\ (j3 + 2) \neq \square$  THEN  
 $tm\_setn\ j3\ 1$

```

ELSE
  []
ENDIF ;;
tm-setn (j3 + 1) 0 ;;
tm-setn (j3 + 2) 0
DONE ;;
tm-cr j1

```

**lemma** *tm-contains-tm*:

**assumes**  $0 < j1$   $j1 \neq j2$   $j3 + 2 < k$   $j1 < j3$   $j2 < j3$  **and**  $k \geq 2$  **and**  $G \geq 5$   
**shows** *turing-machine*  $k$   $G$  (*tm-contains*  $j1$   $j2$   $j3$ )  
*<proof>*

**locale** *turing-machine-contains* =

**fixes**  $j1$   $j2$   $j3$  :: *tapeidx*

**begin**

**definition**  $tmL1 \equiv tm\text{-nextract } | j1 (j3 + 1)$

**definition**  $tmL2 \equiv tmL1$  ;; *tm-equalsn*  $j2 (j3 + 1) (j3 + 2)$

**definition**  $tmI \equiv IF \lambda rs. rs ! (j3 + 2) \neq \square THEN tm\text{-setn } j3 1 ELSE [] ENDIF$

**definition**  $tmL3 \equiv tmL2$  ;;  $tmI$

**definition**  $tmL4 \equiv tmL3$  ;;  $tm\text{-setn } (j3 + 1) 0$

**definition**  $tmL5 \equiv tmL4$  ;;  $tm\text{-setn } (j3 + 2) 0$

**definition**  $tmL \equiv WHILE [] ; \lambda rs. rs ! j1 \neq \square DO tmL5 DONE$

**definition**  $tm2 \equiv tmL$  ;;  $tm\text{-cr } j1$

**lemma** *tm2-eq-tm-contains*:  $tm2 = tm\text{-contains } j1 j2 j3$

*<proof>*

**context**

**fixes**  $tps0$  :: *tape list* **and**  $k$  :: *nat* **and**  $ns$  :: *nat list* **and**  $needle$  :: *nat*  
**assumes**  $jk$ :  $0 < j1$   $j1 \neq j2$   $j3 + 2 < k$   $j1 < j3$   $j2 < j3$   $length\ tps0 = k$   
**and**  $tps0$ :

$tps0 ! j1 = nltape' ns 0$

$tps0 ! j2 = ([needle]_N, 1)$

$tps0 ! j3 = ([0]_N, 1)$

$tps0 ! (j3 + 1) = ([0]_N, 1)$

$tps0 ! (j3 + 2) = ([0]_N, 1)$

**begin**

**definition**  $tpsL$  :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL t \equiv tps0$

$[j1 := nltape' ns t,$

$j3 := ([\exists i < t. ns ! i = needle]_B, 1)]$

**lemma**  $tpsL0$ :  $tpsL 0 = tps0$

*<proof>*

**definition**  $tpsL1$  :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL1 t \equiv tps0$

$[j1 := nltape' ns (Suc t),$

$j3 := ([\exists i < t. ns ! i = needle]_B, 1),$

$j3 + 1 := ([ns ! t]_N, 1)]$

**lemma**  $tmL1$  [*transforms-intros*]:

**assumes**  $ttt = 12 + 2 * nlength (ns ! t)$  **and**  $t < length ns$

**shows** *transforms*  $tmL1 (tpsL t) ttt (tpsL1 t)$

*<proof>*

**definition**  $tpsL2$  :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL2 t \equiv tps0$

$[j1 := nltape' ns (Suc t),$

$j3 := ([\exists i < t. ns ! i = needle]_B, 1),$

$j\mathfrak{3} + 1 := (\lfloor ns ! t \rfloor_N, 1),$   
 $j\mathfrak{3} + 2 := (\lfloor needle = ns ! t \rfloor_B, 1)]$

**lemma** *tmL2* [*transforms-intros*]:

**assumes**  $t\mathfrak{t}t = 12 + 2 * nlength (ns ! t) + (3 * nlength (min\ needle (ns ! t)) + 7)$   
**and**  $t < length\ ns$   
**shows** *transforms tmL2* (*tpsL*  $t$ )  $t\mathfrak{t}t$  (*tpsL2*  $t$ )  
*<proof>*

**definition** *tpsI* :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsI\ t \equiv tps0$   
 $[j1 := nltape' ns (Suc\ t),$   
 $j\mathfrak{3} := (\lfloor \exists i < Suc\ t. ns ! i = needle \rfloor_B, 1),$   
 $j\mathfrak{3} + 1 := (\lfloor ns ! t \rfloor_N, 1),$   
 $j\mathfrak{3} + 2 := (\lfloor needle = ns ! t \rfloor_B, 1)]$

**lemma** *tmI* [*transforms-intros*]:

**assumes**  $t\mathfrak{t}t = 16$  **and**  $t < length\ ns$   
**shows** *transforms tmI* (*tpsL2*  $t$ )  $t\mathfrak{t}t$  (*tpsI*  $t$ )  
*<proof>*

**lemma** *tmL3* [*transforms-intros*]:

**assumes**  $t\mathfrak{t}t = 28 + 2 * nlength (ns ! t) + (3 * nlength (min\ needle (ns ! t)) + 7)$   
**and**  $t < length\ ns$   
**shows** *transforms tmL3* (*tpsL*  $t$ )  $t\mathfrak{t}t$  (*tpsI*  $t$ )  
*<proof>*

**definition** *tpsL4* :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL4\ t \equiv tps0$   
 $[j1 := nltape' ns (Suc\ t),$   
 $j\mathfrak{3} := (\lfloor \exists i < Suc\ t. ns ! i = needle \rfloor_B, 1),$   
 $j\mathfrak{3} + 1 := (\lfloor 0 \rfloor_N, 1),$   
 $j\mathfrak{3} + 2 := (\lfloor needle = ns ! t \rfloor_B, 1)]$

**lemma** *tmL4* [*transforms-intros*]:

**assumes**  $t\mathfrak{t}t = 38 + 4 * nlength (ns ! t) + (3 * nlength (min\ needle (ns ! t)) + 7)$   
**and**  $t < length\ ns$   
**shows** *transforms tmL4* (*tpsL*  $t$ )  $t\mathfrak{t}t$  (*tpsL4*  $t$ )  
*<proof>*

**definition** *tpsL5* :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL5\ t \equiv tps0$   
 $[j1 := nltape' ns (Suc\ t),$   
 $j\mathfrak{3} := (\lfloor \exists i < Suc\ t. ns ! i = needle \rfloor_B, 1),$   
 $j\mathfrak{3} + 1 := (\lfloor 0 \rfloor_N, 1),$   
 $j\mathfrak{3} + 2 := (\lfloor 0 \rfloor_N, 1)]$

**lemma** *tmL5*:

**assumes**  $t\mathfrak{t}t = 48 + 4 * nlength (ns ! t) + (3 * nlength (min\ needle (ns ! t)) + 7) +$   
 $2 * nlength (if\ needle = ns ! t\ then\ 1\ else\ 0)$   
**and**  $t < length\ ns$   
**shows** *transforms tmL5* (*tpsL*  $t$ )  $t\mathfrak{t}t$  (*tpsL5*  $t$ )  
*<proof>*

**definition** *tpsL5'* :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL5'\ t \equiv tps0$   
 $[j1 := nltape' ns (Suc\ t),$   
 $j\mathfrak{3} := (\lfloor \exists i < Suc\ t. ns ! i = needle \rfloor_B, 1)]$

**lemma** *tpsL5'*: *tpsL5'*  $t = tpsL5\ t$

*<proof>*

**lemma** *tmL5'*:

**assumes**  $t\mathfrak{t}t = 57 + 4 * nlength (ns ! t) + 3 * nlength (min\ needle (ns ! t))$

**and**  $t < \text{length } ns$   
**shows** *transforms tmL5* (*tpsL*  $t$ ) *ttt* (*tpsL5'*  $t$ )  
 ⟨*proof*⟩

**lemma** *tmL5''* [*transforms-intros*]:  
**assumes**  $ttt = 57 + 7 * \text{nlength } ns$  **and**  $t < \text{length } ns$   
**shows** *transforms tmL5* (*tpsL*  $t$ ) *ttt* (*tpsL* (*Suc*  $t$ ))  
 ⟨*proof*⟩

**lemma** *tmL* [*transforms-intros*]:  
**assumes**  $ttt = \text{length } ns * (59 + 7 * \text{nlength } ns) + 1$   
**shows** *transforms tmL* (*tpsL*  $0$ ) *ttt* (*tpsL* ( $\text{length } ns$ ))  
 ⟨*proof*⟩

**definition** *tps2* :: *tape list* **where**  
 $tps2 \equiv tps0$   
 $[j1 := \text{nltape}' ns 0,$   
 $j3 := (\exists i < \text{length } ns. ns ! i = \text{needle}]_B, 1)]$

**lemma** *tm2*:  
**assumes**  $ttt = \text{length } ns * (59 + 7 * \text{nlength } ns) + \text{nlength } ns + 4$   
**shows** *transforms tm2* (*tpsL*  $0$ ) *ttt* *tps2*  
 ⟨*proof*⟩

**definition** *tps2'* :: *tape list* **where**  
 $tps2' \equiv tps0$   
 $[j3 := (\text{needle} \in \text{set } ns)_B, 1)]$

**lemma** *tm2'*:  
**assumes**  $ttt = 67 * \text{nlength } ns \wedge 2 + 4$   
**shows** *transforms tm2* *tps0* *ttt* *tps2'*  
 ⟨*proof*⟩

**end**

**end**

**lemma** *transforms-tm-containsI* [*transforms-intros*]:  
**fixes**  $j1 j2 j3$  :: *tapeidx*  
**fixes** *tps* *tps'* :: *tape list* **and**  $ttt k \text{needle}$  :: *nat* **and**  $ns$  :: *nat list*  
**assumes**  $0 < j1 j1 \neq j2 j3 + 2 < k j1 < j3 j2 < j3 \text{length } tps = k$   
**assumes**  
 $tps ! j1 = \text{nltape}' ns 0$   
 $tps ! j2 = (\text{needle}]_N, 1)$   
 $tps ! j3 = ([0]_N, 1)$   
 $tps ! (j3 + 1) = ([0]_N, 1)$   
 $tps ! (j3 + 2) = ([0]_N, 1)$   
**assumes**  $ttt = 67 * \text{nlength } ns \wedge 2 + 4$   
**assumes**  $tps' = tps$   
 $[j3 := (\text{needle} \in \text{set } ns)_B, 1)]$   
**shows** *transforms (tm-contains*  $j1 j2 j3$ ) *tps* *ttt* *tps'*  
 ⟨*proof*⟩

## 2.8.8 Creating lists of consecutive numbers

The next TM accepts a number *start* on tape  $j_1$  and a number *delta* on tape  $j_2$ . It outputs the list  $[start, \dots, start + delta - 1]$  on tape  $j_3$ .

**definition** *tm-range* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**  
 $tm\text{-range } j1 j2 j3 \equiv$   
 $tm\text{-copyn } j1 (j3 + 2) ;;$   
 $tm\text{-copyn } j2 (j3 + 1) ;;$   
 $WHILE [] ; \lambda rs. rs ! (j3 + 1) \neq \square DO$   
 $tm\text{-append } j3 (j3 + 2) ;;$

```

  tm-incr (j3 + 2) ;;
  tm-decr (j3 + 1)
  DONE ;;
  tm-setn (j3 + 2) 0 ;;
  tm-cr j3

```

**lemma** *tm-range-tm*:

**assumes**  $k \geq j3 + 3$  **and**  $G \geq 5$  **and**  $j1 \neq j2$  **and**  $0 < j1$  **and**  $0 < j2$  **and**  $j1 < j3$  **and**  $j2 < j3$   
**shows** *turing-machine*  $k$   $G$  (*tm-range*  $j1$   $j2$   $j3$ )  
 ⟨*proof*⟩

**locale** *turing-machine-range* =

**fixes**  $j1$   $j2$   $j3$  :: *tapeidx*

**begin**

**definition**  $tm1 \equiv tm\text{-copyn } j1 (j3 + 2)$

**definition**  $tm2 \equiv tm1$  ;; *tm-copyn*  $j2 (j3 + 1)$

**definition**  $tmB1 \equiv tm\text{-append } j3 (j3 + 2)$

**definition**  $tmB2 \equiv tmB1$  ;; *tm-incr*  $(j3 + 2)$

**definition**  $tmB3 \equiv tmB2$  ;; *tm-decr*  $(j3 + 1)$

**definition**  $tmL \equiv WHILE [] ; \lambda rs. rs ! (j3 + 1) \neq [] DO tmB3$  *DONE*

**definition**  $tm3 \equiv tm2$  ;;  $tmL$

**definition**  $tm4 \equiv tm3$  ;; *tm-setn*  $(j3 + 2)$  0

**definition**  $tm5 \equiv tm4$  ;; *tm-cr*  $j3$

**lemma** *tm5-eq-tm-range*:  $tm5 = tm\text{-range } j1$   $j2$   $j3$

⟨*proof*⟩

**context**

**fixes**  $tps0$  :: *tape list* **and**  $k$  *start delta* :: *nat*

**assumes**  $jk$ :  $k \geq j3 + 3$   $j1 \neq j2$   $0 < j1$   $0 < j2$   $0 < j3$   $j1 < j3$   $j2 < j3$   $\text{length } tps0 = k$

**and**  $tps0$ :

$tps0 ! j1 = ([start]_N, 1)$

$tps0 ! j2 = ([delta]_N, 1)$

$tps0 ! j3 = ([ ]_{NL}, 1)$

$tps0 ! (j3 + 1) = ([0]_N, 1)$

$tps0 ! (j3 + 2) = ([0]_N, 1)$

**begin**

**definition**  $tps1 \equiv tps0$

$[j3 + 2 := ([start]_N, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:

**assumes**  $ttt = 14 + 3 * \text{nlength } start$

**shows** *transforms*  $tm1$   $tps0$   $ttt$   $tps1$

⟨*proof*⟩

**definition**  $tps2 \equiv tps0$

$[j3 + 2 := ([start]_N, 1),$

$j3 + 1 := ([delta]_N, 1)]$

**lemma**  $tm2$  [*transforms-intros*]:

**assumes**  $ttt = 28 + 3 * \text{nlength } start + 3 * \text{nlength } delta$

**shows** *transforms*  $tm2$   $tps0$   $ttt$   $tps2$

⟨*proof*⟩

**definition**  $tpsL$   $t \equiv tps0$

$[j3 + 2 := ([start + t]_N, 1),$

$j3 + 1 := ([delta - t]_N, 1),$

$j3 := \text{nltape } [start..<start + t]$

**lemma**  $tpsL\text{-eq-tps2}$ :  $tpsL$  0 =  $tps2$

⟨*proof*⟩

**definition**  $tpsB1\ t \equiv tps0$

$[j3 + 2 := (\lfloor start + t \rfloor_N, 1),$   
 $j3 + 1 := (\lfloor delta - t \rfloor_N, 1),$   
 $j3 := nltape ([start..<start + t] @ [start + t])]$

**lemma**  $tmB1$  [*transforms-intros*]:

**assumes**  $ttt = 6 + 2 * nlength (start + t)$   
**shows**  $transforms\ tmB1\ (tpsL\ t)\ ttt\ (tpsB1\ t)$   
*<proof>*

**definition**  $tpsB2\ t \equiv tps0$

$[j3 + 2 := (\lfloor Suc (start + t) \rfloor_N, 1),$   
 $j3 + 1 := (\lfloor delta - t \rfloor_N, 1),$   
 $j3 := nltape ([start..<start + t] @ [start + t])]$

**lemma**  $tmB2$  [*transforms-intros*]:

**assumes**  $ttt = 11 + 4 * nlength (start + t)$   
**shows**  $transforms\ tmB2\ (tpsL\ t)\ ttt\ (tpsB2\ t)$   
*<proof>*

**definition**  $tpsB3\ t \equiv tps0$

$[j3 + 2 := (\lfloor Suc (start + t) \rfloor_N, 1),$   
 $j3 + 1 := (\lfloor delta - t - 1 \rfloor_N, 1),$   
 $j3 := nltape ([start..<start + t] @ [start + t])]$

**lemma**  $tmB3$ :

**assumes**  $ttt = 19 + 4 * nlength (start + t) + 2 * nlength (delta - t)$   
**shows**  $transforms\ tmB3\ (tpsL\ t)\ ttt\ (tpsB3\ t)$   
*<proof>*

**lemma**  $tpsB3$ :  $tpsB3\ t \equiv tpsL (Suc\ t)$

*<proof>*

**lemma**  $tmB3'$  [*transforms-intros*]:

**assumes**  $ttt = 19 + 6 * nlength (start + delta)$  **and**  $t < delta$   
**shows**  $transforms\ tmB3\ (tpsL\ t)\ ttt\ (tpsL (Suc\ t))$   
*<proof>*

**lemma**  $tmL$ :

**assumes**  $ttt = delta * (21 + 6 * nlength (start + delta)) + 1$   
**shows**  $transforms\ tmL\ (tpsL\ 0)\ ttt\ (tpsL\ delta)$   
*<proof>*

**lemma**  $tmL'$  [*transforms-intros*]:

**assumes**  $ttt = delta * (21 + 6 * nlength (start + delta)) + 1$   
**shows**  $transforms\ tmL\ tps2\ ttt\ (tpsL\ delta)$   
*<proof>*

**definition**  $tps3 \equiv tps0$

$[j3 + 2 := (\lfloor start + delta \rfloor_N, 1),$   
 $j3 := nltape [start..<start + delta]]$

**lemma**  $tpsL\ eq\ tps3$ :  $tpsL\ delta = tps3$

*<proof>*

**lemma**  $tm3$ :

**assumes**  $ttt = 29 + 3 * nlength\ start + 3 * nlength\ delta + delta * (21 + 6 * nlength (start + delta))$   
**shows**  $transforms\ tm3\ tps0\ ttt\ (tpsL\ delta)$   
*<proof>*

**lemma**  $tm3'$  [*transforms-intros*]:

**assumes**  $ttt = 29 + 3 * nlength\ start + 3 * nlength\ delta + delta * (21 + 6 * nlength (start + delta))$

**shows** *transforms tm3 tps0 ttt tps3*  
 ⟨*proof*⟩

**definition** *tps4*  $\equiv$  *tps0*  
 $[j3 := \text{nltape } [start..<start + delta]]$

**lemma** *tm4*:  
**assumes**  $ttt = 39 + 3 * \text{nlength } start + 3 * \text{nlength } delta + delta * (21 + 6 * \text{nlength } (start + delta)) + 2 * \text{nlength } (start + delta)$   
**shows** *transforms tm4 tps0 ttt tps4*  
 ⟨*proof*⟩

**lemma** *tm4'* [*transforms-intros*]:  
**assumes**  $ttt = \text{Suc } delta * (39 + 8 * \text{nlength } (start + delta))$   
**shows** *transforms tm4 tps0 ttt tps4*  
 ⟨*proof*⟩

**definition** *tps5*  $\equiv$  *tps0*  
 $[j3 := (\llbracket start..<start + delta \rrbracket_{NL}, 1)]$

**lemma** *tm5*:  
**assumes**  $ttt = \text{Suc } delta * (39 + 8 * \text{nlength } (start + delta)) + \text{Suc } (\text{Suc } (\text{Suc } (\text{nlength } [start..<start + delta])))$   
**shows** *transforms tm5 tps0 ttt tps5*  
 ⟨*proof*⟩

**lemma** *tm5'*:  
**assumes**  $ttt = \text{Suc } delta * (43 + 9 * \text{nlength } (start + delta))$   
**shows** *transforms tm5 tps0 ttt tps5*  
 ⟨*proof*⟩

**end**

**end**

**lemma** *transforms-tm-rangeI* [*transforms-intros*]:  
**fixes**  $j1 j2 j3 :: \text{tapeidx}$   
**fixes**  $tps tps' :: \text{tape list}$  **and**  $k start delta :: \text{nat}$   
**assumes**  $k \geq j3 + 3$   $j1 \neq j2$   $0 < j1$   $0 < j2$   $j1 < j3$   $j2 < j3$   $\text{length } tps = k$   
**assumes**  
 $tps ! j1 = (\llbracket start \rrbracket_N, 1)$   
 $tps ! j2 = (\llbracket delta \rrbracket_N, 1)$   
 $tps ! j3 = (\llbracket \square \rrbracket_{NL}, 1)$   
 $tps ! (j3 + 1) = (\llbracket 0 \rrbracket_N, 1)$   
 $tps ! (j3 + 2) = (\llbracket 0 \rrbracket_N, 1)$   
**assumes**  $ttt = \text{Suc } delta * (43 + 9 * \text{nlength } (start + delta))$   
**assumes**  $tps' = tps$   
 $[j3 := (\llbracket start..<start + delta \rrbracket_{NL}, 1)]$   
**shows** *transforms (tm-range j1 j2 j3) tps ttt tps'*  
 ⟨*proof*⟩

## 2.8.9 Creating singleton lists

The next Turing machine appends the symbol | to the symbols on tape  $j$ . Thus it turns a number into a singleton list containing this number.

**definition** *tm-to-list*  $:: \text{tapeidx} \Rightarrow \text{machine}$  **where**  
 $tm\text{-to-list } j \equiv$   
 $tm\text{-right-until } j \{ \square \} ;;$   
 $tm\text{-write } j \mid ;;$   
 $tm\text{-cr } j$

**lemma** *tm-to-list-tm*:  
**assumes**  $0 < j$  **and**  $j < k$  **and**  $G \geq 5$

```

shows turing-machine k G (tm-to-list j)
  ⟨proof⟩

locale turing-machine-to-list =
  fixes j :: tapeidx
begin

definition tm1 ≡ tm-right-until j {□}
definition tm2 ≡ tm1 ;; tm-write j |
definition tm3 ≡ tm2 ;; tm-cr j

lemma tm3-eq-tm-to-list: tm3 = tm-to-list j
  ⟨proof⟩

context
  fixes tps0 :: tape list and k n :: nat
  assumes jk: 0 < j j < k length tps0 = k
  and tps0: tps0 ! j = (⌊n⌋N, 1)
begin

definition tps1 ≡ tps0[j := (⌊n⌋N, Suc (nlength n))]

lemma tm1 [transforms-intros]:
  assumes ttt = Suc (nlength n)
  shows transforms tm1 tps0 ttt tps1
  ⟨proof⟩

definition tps2 ≡ tps0[j := (⌊n⌋NL, Suc (nlength n))]

lemma tm2 [transforms-intros]:
  assumes ttt = Suc (Suc (nlength n))
  shows transforms tm2 tps0 ttt tps2
  ⟨proof⟩

definition tps3 ≡ tps0[j := (⌊n⌋NL, 1)]

lemma tm3:
  assumes ttt = 5 + 2 * nlength n
  shows transforms tm3 tps0 ttt tps3
  ⟨proof⟩

end

end

lemma transforms-tm-to-listI [transforms-intros]:
  fixes j :: tapeidx
  fixes tps tps' :: tape list and ttt k n :: nat
  assumes 0 < j j < k length tps = k
  assumes tps ! j = (⌊n⌋N, 1)
  assumes ttt = 5 + 2 * nlength n
  assumes tps' = tps[j := (⌊n⌋NL, 1)]
  shows transforms (tm-to-list j) tps ttt tps'
  ⟨proof⟩

```

### 2.8.10 Extending with a list

The next Turing machine extends the list on tape  $j_1$  with the list on tape  $j_2$ . We assume that the tape head on  $j_1$  is already at the end of the list.

```

definition tm-extend :: tapeidx ⇒ tapeidx ⇒ machine where
  tm-extend j1 j2 ≡ tm-cp-until j2 j1 {□} ;; tm-cr j2

```

```

lemma tm-extend-tm:

```

**assumes**  $0 < j1$  **and**  $G \geq 4$  **and**  $j1 < k$  **and**  $j2 < k$   
**shows** *turing-machine*  $k$   $G$  (*tm-extend*  $j1$   $j2$ )  
*<proof>*

**locale** *turing-machine-extend* =  
**fixes**  $j1$   $j2$  :: *tapeidx*  
**begin**

**definition**  $tm1 \equiv tm\text{-}cp\text{-}until$   $j2$   $j1$   $\{\square\}$

**definition**  $tm2 \equiv tm1$  ;; *tm-cr*  $j2$

**lemma** *tm2-eq-tm-extend*:  $tm2 = tm\text{-}extend$   $j1$   $j2$   
*<proof>*

**context**

**fixes**  $tps0$  :: *tape list* **and**  $k$  :: *nat* **and**  $ns1$   $ns2$  :: *nat list*  
**assumes**  $jk$ :  $0 < j1$   $j1 < k$   $j2 < k$   $j1 \neq j2$  *length*  $tps0 = k$   
**assumes**  $tps0$ :

$tps0 ! j1 = nltape$   $ns1$   
 $tps0 ! j2 = ([ns2]_{NL}, 1)$

**begin**

**definition**  $tps1 \equiv tps0$

$[j1 := nltape$  ( $ns1 @ ns2$ ),  
 $j2 := nltape$   $ns2]$

**lemma**  $tm1$  [*transforms-intros*]:  
**assumes**  $ttt = Suc$  (*nlength*  $ns2$ )  
**shows** *transforms*  $tm1$   $tps0$   $ttt$   $tps1$   
*<proof>*

**definition**  $tps2 \equiv tps0[j1 := nltape$  ( $ns1 @ ns2$ )]

**lemma**  $tm2$ :  
**assumes**  $ttt = 4 + 2 * nlength$   $ns2$   
**shows** *transforms*  $tm2$   $tps0$   $ttt$   $tps2$   
*<proof>*

**end**

**end**

**lemma** *transforms-tm-extendI* [*transforms-intros*]:  
**fixes**  $j1$   $j2$  :: *tapeidx*  
**fixes**  $tps$   $tps'$  :: *tape list* **and**  $k$  :: *nat* **and**  $ns1$   $ns2$  :: *nat list*  
**assumes**  $0 < j1$   $j1 < k$   $j2 < k$   $j1 \neq j2$  *length*  $tps = k$   
**assumes**  
 $tps ! j1 = nltape$   $ns1$   
 $tps ! j2 = ([ns2]_{NL}, 1)$   
**assumes**  $ttt = 4 + 2 * nlength$   $ns2$   
**assumes**  $tps' = tps[j1 := nltape$  ( $ns1 @ ns2$ )]  
**shows** *transforms* (*tm-extend*  $j1$   $j2$ )  $tps$   $ttt$   $tps'$   
*<proof>*

An enhanced version of the previous Turing machine, the next one erases the list on tape  $j_2$  after appending it to tape  $j_1$ .

**definition** *tm-extend-erase* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**  
 $tm\text{-}extend\text{-}erase$   $j1$   $j2 \equiv tm\text{-}extend$   $j1$   $j2$  ;; *tm-erase-cr*  $j2$

**lemma** *tm-extend-erase-tm*:  
**assumes**  $0 < j1$  **and**  $0 < j2$  **and**  $G \geq 4$  **and**  $j1 < k$  **and**  $j2 < k$   
**shows** *turing-machine*  $k$   $G$  (*tm-extend-erase*  $j1$   $j2$ )  
*<proof>*

**lemma** *transforms-tm-extend-eraseI* [*transforms-intros*]:  
**fixes**  $j1\ j2 :: \text{tapeid}x$   
**fixes**  $tps\ tps' :: \text{tape list}$  **and**  $k :: \text{nat}$  **and**  $ns1\ ns2 :: \text{nat list}$   
**assumes**  $0 < j1\ j1 < k\ j2 < k\ j1 \neq j2\ 0 < j2\ \text{length } tps = k$   
**assumes**  
 $tps ! j1 = \text{nltape } ns1$   
 $tps ! j2 = ([ns2]_{NL}, 1)$   
**assumes**  $ttt = 11 + 4 * \text{nlength } ns2$   
**assumes**  $tps' = tps$   
 $[j1 := \text{nltape } (ns1 @ ns2),$   
 $j2 := ([[]], 1)]$   
**shows** *transforms (tm-extend-erase j1 j2) tps ttt tps'*  
*<proof>*

## 2.9 Lists of lists of numbers

In this section we introduce a representation for lists of lists of numbers as symbol sequences over the quaternary alphabet  $\mathbf{01}\#\#$  and devise Turing machines for the few operations on such lists that we need. A tape containing such representations corresponds to a variable of type *nat list list*. A tape in the start configuration corresponds to the empty list of lists of numbers.

Many proofs in this section are copied from the previous section with minor modifications, mostly replacing the symbol  $|$  with  $\#$ .

### 2.9.1 Representation as symbol sequence

We apply the same principle as for representing lists of numbers. To represent a list of lists of numbers, every element, which is now a list of numbers, is terminated by the symbol  $\#$ . In this way the empty symbol sequence represents the empty list of lists of numbers. The list  $[[]]$  containing only an empty list is represented by  $\#$  and the list  $[[0]]$  containing only a list containing only a 0 is represented by  $\#$ . As another example, the list  $[[14, 0, 0, 7], [], [0], [25]]$  is represented by  $\mathbf{0111}\#|111\#\#|10011\#$ . The number of  $\#$  symbols equals the number of elements in the list.

**definition** *numlistlist*  $:: \text{nat list list} \Rightarrow \text{symbol list}$  **where**  
 $\text{numlistlist } nss \equiv \text{concat } (\text{map } (\lambda ns. \text{numlist } ns @ [\#])\ nss)$

**lemma** *numlistlist-Nil*:  $\text{numlistlist } [] = []$   
*<proof>*

**proposition** *numlistlist [[]]*  $= [\#]$   
*<proof>*

**lemma** *proper-symbols-numlistlist*: *proper-symbols (numlistlist nss)*  
*<proof>*

**lemma** *symbols-lt-append*:  
**fixes**  $xs\ ys :: \text{symbol list}$  **and**  $z :: \text{symbol}$   
**assumes** *symbols-lt z xs* **and** *symbols-lt z ys*  
**shows** *symbols-lt z (xs @ ys)*  
*<proof>*

**lemma** *symbols-lt-numlistlist*:  
**assumes**  $H \geq 6$   
**shows** *symbols-lt H (numlistlist nss)*  
*<proof>*

**lemma** *symbols-lt-prefix-eq*:  
**assumes**  $(x @ [\#]) @ xs = (y @ [\#]) @ ys$  **and** *symbols-lt 5 x* **and** *symbols-lt 5 y*  
**shows**  $x = y$   
*<proof>*

**lemma numlistlist-inj:**  $\text{numlistlist } ns1 = \text{numlistlist } ns2 \implies ns1 = ns2$   
 ⟨proof⟩

**lemma numlistlist-append:**  $\text{numlistlist } (xs @ ys) = \text{numlistlist } xs @ \text{numlistlist } ys$   
 ⟨proof⟩

Similar to  $[-]_N$  and  $[-]_{NL}$ , the tape contents for a list of list of numbers:

**definition nllcontents** ::  $\text{nat list list} \Rightarrow (\text{nat} \Rightarrow \text{symbol}) (\langle [-]_{NLL} \rangle)$  **where**  
 $[nss]_{NLL} \equiv [\text{numlistlist } nss]$

**lemma clean-tape-nllcontents:**  $\text{clean-tape } ([ns]_{NLL}, i)$   
 ⟨proof⟩

**lemma nllcontents-Nil:**  $[\square]_{NLL} = [\square]$   
 ⟨proof⟩

Similar to  $nlength$  and  $nllength$ , the length of the representation of a list of lists of numbers:

**definition nlllength** ::  $\text{nat list list} \Rightarrow \text{nat}$  **where**  
 $nlllength \ nss \equiv \text{length } (\text{numlistlist } nss)$

**lemma nlllength:**  $nlllength \ nss = (\sum ns \leftarrow nss. \text{Suc } (nlength \ ns))$   
 ⟨proof⟩

**lemma nlllength-Nil [simp]:**  $nlllength \ [] = 0$   
 ⟨proof⟩

**lemma nlllength-Cons:**  $nlllength \ (ns \# nss) = \text{Suc } (nlength \ ns) + nlllength \ nss$   
 ⟨proof⟩

**lemma length-le-nlllength:**  $\text{length } nss \leq nlllength \ nss$   
 ⟨proof⟩

**lemma member-le-nlllength-1:**  $ns \in \text{set } nss \implies nlength \ ns \leq nlllength \ nss - 1$   
 ⟨proof⟩

**lemma nlllength-take:**  
**assumes**  $i < \text{length } nss$   
**shows**  $nlllength \ (\text{take } i \ nss) + nlength \ (nss ! i) < nlllength \ nss$   
 ⟨proof⟩

**lemma take-drop-numlistlist:**  
**assumes**  $i < \text{length } ns$   
**shows**  $\text{take } (\text{Suc } (nlength \ (ns ! i))) \ (\text{drop } (nllength \ (\text{take } i \ ns)) \ (\text{numlistlist } ns)) = \text{numlist } (ns ! i) @ [\#]$   
 ⟨proof⟩

**corollary take-drop-numlistlist':**  
**assumes**  $i < \text{length } ns$   
**shows**  $\text{take } (nlength \ (ns ! i)) \ (\text{drop } (nllength \ (\text{take } i \ ns)) \ (\text{numlistlist } ns)) = \text{numlist } (ns ! i)$   
 ⟨proof⟩

**corollary numlistlist-take-at-term:**  
**assumes**  $i < \text{length } ns$   
**shows**  $\text{numlistlist } ns ! (nllength \ (\text{take } i \ ns) + nlength \ (ns ! i)) = \#$   
 ⟨proof⟩

**lemma nlllength-take-Suc:**  
**assumes**  $i < \text{length } ns$   
**shows**  $nlllength \ (\text{take } i \ ns) + \text{Suc } (nlength \ (ns ! i)) = nlllength \ (\text{take } (\text{Suc } i) \ ns)$   
 ⟨proof⟩

**lemma numlistlist-take-at:**  
**assumes**  $i < \text{length } ns$  **and**  $j < nlength \ (ns ! i)$

**shows**  $\text{numlistlist } ns ! (\text{nllength } (\text{take } i \text{ } ns) + j) = \text{numlist } (ns ! i) ! j$   
 ⟨proof⟩

**lemma** *nllcontents-rneigh-5*:

**assumes**  $i < \text{length } ns$

**shows**  $\text{rneigh } (\lfloor ns \rfloor_{NLL}, \text{Suc } (\text{nllength } (\text{take } i \text{ } ns))) \{ \# \} (\text{nllength } (ns ! i))$

⟨proof⟩

A tape containing a list of lists of numbers, with the tape head after all the symbols:

**abbreviation**  $\text{nlltape} :: \text{nat list list} \Rightarrow \text{tape}$  **where**

$\text{nlltape } ns \equiv (\lfloor ns \rfloor_{NLL}, \text{Suc } (\text{nlllength } ns))$

Like before but with the tape head or at the beginning of the  $i$ -th list element:

**abbreviation**  $\text{nlltape}' :: \text{nat list list} \Rightarrow \text{nat} \Rightarrow \text{tape}$  **where**

$\text{nlltape}' ns i \equiv (\lfloor ns \rfloor_{NLL}, \text{Suc } (\text{nlllength } (\text{take } i \text{ } ns)))$

**lemma** *nlltape'-0*:  $\text{nlltape}' ns 0 = (\lfloor ns \rfloor_{NLL}, 1)$

⟨proof⟩

**lemma** *nlltape'-tape-read*:  $|\cdot| (\text{nlltape}' nss i) = 0 \iff i \geq \text{length } nss$

⟨proof⟩

## 2.9.2 Appending an element

The next Turing machine is very similar to *tm-append*, just with terminator symbol  $\#$  instead of  $|$ . It appends a list of numbers given on tape  $j_2$  to the list of lists of numbers on tape  $j_1$ .

**definition** *tm-appendl* ::  $\text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**

$\text{tm-appendl } j1 j2 \equiv$

$\text{tm-right-until } j1 \{ \square \} ;;$

$\text{tm-cp-until } j2 j1 \{ \square \} ;;$

$\text{tm-cr } j2 ;;$

$\text{tm-char } j1 \#$

**lemma** *tm-appendl-tm*:

**assumes**  $0 < j1$  **and**  $G \geq 6$  **and**  $j1 < k$  **and**  $j2 < k$

**shows**  $\text{turing-machine } k G (\text{tm-appendl } j1 j2)$

⟨proof⟩

**locale** *turing-machine-appendl* =

**fixes**  $j1 j2 :: \text{tapeidx}$

**begin**

**definition**  $\text{tm1} \equiv \text{tm-right-until } j1 \{ \square \}$

**definition**  $\text{tm2} \equiv \text{tm1} ;; \text{tm-cp-until } j2 j1 \{ \square \}$

**definition**  $\text{tm3} \equiv \text{tm2} ;; \text{tm-cr } j2$

**definition**  $\text{tm4} \equiv \text{tm3} ;; \text{tm-char } j1 \#$

**lemma** *tm4-eq-tm-append*:  $\text{tm4} = \text{tm-appendl } j1 j2$

⟨proof⟩

**context**

**fixes**  $\text{tps0} :: \text{tape list}$  **and**  $k i1 :: \text{nat}$  **and**  $ns :: \text{nat list}$  **and**  $nss :: \text{nat list list}$

**assumes**  $\text{jk}$ :  $\text{length } \text{tps0} = k$   $j1 < k$   $j2 < k$   $j1 \neq j2$   $0 < j1$

**and**  $i1$ :  $i1 \leq \text{Suc } (\text{nllength } nss)$

**assumes**  $\text{tps0}$ :

$\text{tps0} ! j1 = (\lfloor nss \rfloor_{NLL}, i1)$

$\text{tps0} ! j2 = (\lfloor ns \rfloor_{NL}, 1)$

**begin**

**definition**  $\text{tps1} \equiv \text{tps0}[j1 := \text{nlltape } nss]$

**lemma** *tm1* [*transforms-intros*]:

**assumes**  $ttt = \text{Suc} (\text{Suc} (\text{nllength } nss) - i1)$   
**shows**  $\text{transforms } tm1 \ tps0 \ ttt \ tps1$   
 $\langle \text{proof} \rangle$

**definition**  $tps2 \equiv tps0$

$[j1 := (\lfloor \text{numlistlist } nss \ @ \ \text{numlist } ns \rfloor, \text{Suc} (\text{nllength } nss) + \text{nllength } ns),$   
 $j2 := (\lfloor ns \rfloor_{NL}, \text{Suc} (\text{nllength } ns))]$

**lemma**  $tm2$  [*transforms-intros*]:

**assumes**  $ttt = 3 + \text{nllength } nss - i1 + \text{nllength } ns$   
**shows**  $\text{transforms } tm2 \ tps0 \ ttt \ tps2$   
 $\langle \text{proof} \rangle$

**definition**  $tps3 \equiv tps0$

$[j1 := (\lfloor \text{numlistlist } nss \ @ \ \text{numlist } ns \rfloor, \text{Suc} (\text{nllength } nss) + \text{nllength } ns)]$

**lemma**  $tm3$  [*transforms-intros*]:

**assumes**  $ttt = 6 + \text{nllength } nss - i1 + 2 * \text{nllength } ns$   
**shows**  $\text{transforms } tm3 \ tps0 \ ttt \ tps3$   
 $\langle \text{proof} \rangle$

**definition**  $tps4 \equiv tps0$

$[j1 := (\lfloor \text{numlistlist } (nss \ @ \ [ns]) \rfloor, \text{Suc} (\text{nllength } (nss \ @ \ [ns])))]$

**lemma**  $tm4$ :

**assumes**  $ttt = 7 + \text{nllength } nss - i1 + 2 * \text{nllength } ns$   
**shows**  $\text{transforms } tm4 \ tps0 \ ttt \ tps4$   
 $\langle \text{proof} \rangle$

**end**

**end**

**lemma** *transforms-tm-appendII* [*transforms-intros*]:

**fixes**  $j1 \ j2 :: \text{tapeidx}$

**fixes**  $tps \ tps' :: \text{tape list}$  **and**  $ttt \ k \ i1 :: \text{nat}$  **and**  $ns :: \text{nat list}$  **and**  $nss :: \text{nat list list}$

**assumes**  $0 < j1$

**assumes**  $\text{length } tps = k \ j1 < k \ j2 < k \ j1 \neq j2$

**and**  $i1 \leq \text{Suc} (\text{nllength } nss)$

**assumes**

$tps \ ! \ j1 = (\lfloor nss \rfloor_{NLL}, i1)$

$tps \ ! \ j2 = (\lfloor ns \rfloor_{NL}, 1)$

**assumes**  $ttt = 7 + \text{nllength } nss - i1 + 2 * \text{nllength } ns$

**assumes**  $tps' = tps$

$[j1 := \text{nlltape } (nss \ @ \ [ns])]$

**shows**  $\text{transforms } (tm\text{-appendI } j1 \ j2) \ tps \ ttt \ tps'$

$\langle \text{proof} \rangle$

### 2.9.3 Extending with a list

The next Turing machine extends a list of lists of numbers with another list of lists of numbers. It is in fact the same TM as for extending a list of numbers because in both cases extending means simply copying the contents from one tape to another. We introduce a new name for this TM and express the semantics in terms of lists of lists of numbers. The proof is almost the same except with *nllength* replaced by *nlllength* and so on.

**definition**  $tm\text{-extendI} :: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**

$tm\text{-extendI} \equiv tm\text{-extend}$

**lemma**  $tm\text{-extendI-tm}$ :

**assumes**  $0 < j1$  **and**  $G \geq 4$  **and**  $j1 < k$  **and**  $j2 < k$

**shows**  $\text{turing-machine } k \ G \ (tm\text{-extendI } j1 \ j2)$

$\langle \text{proof} \rangle$

```

locale turing-machine-extendl =
  fixes j1 j2 :: tapeidx
begin

definition tm1  $\equiv$  tm-cp-until j2 j1 { $\square$ }
definition tm2  $\equiv$  tm1 ;; tm-cr j2

lemma tm2-eq-tm-extendl: tm2 = tm-extendl j1 j2
   $\langle$ proof $\rangle$ 

context
  fixes tps0 :: tape list and k :: nat and nss1 nss2 :: nat list list
  assumes jk:  $0 < j1$   $j1 < k$   $j2 < k$   $j1 \neq j2$  length tps0 = k
  assumes tps0:
    tps0 ! j1 = nlltape nss1
    tps0 ! j2 = ( $\lfloor$ nss2 $\rfloor_{NLL}$ , 1)
begin

definition tps1  $\equiv$  tps0
  [j1 := nlltape (nss1 @ nss2),
   j2 := nlltape nss2]

lemma tm1 [transforms-intros]:
  assumes ttt = Suc (nlllength nss2)
  shows transforms tm1 tps0 ttt tps1
   $\langle$ proof $\rangle$ 

definition tps2  $\equiv$  tps0[j1 := nlltape (nss1 @ nss2)]

lemma tm2:
  assumes ttt =  $4 + 2 * \text{nlllength } nss2$ 
  shows transforms tm2 tps0 ttt tps2
   $\langle$ proof $\rangle$ 

end

end

lemma transforms-tm-extendlI [transforms-intros]:
  fixes j1 j2 :: tapeidx
  fixes tps tps' :: tape list and k :: nat and nss1 nss2 :: nat list list
  assumes  $0 < j1$   $j1 < k$   $j2 < k$   $j1 \neq j2$  length tps = k
  assumes
    tps ! j1 = nlltape nss1
    tps ! j2 = ( $\lfloor$ nss2 $\rfloor_{NLL}$ , 1)
  assumes ttt =  $4 + 2 * \text{nlllength } nss2$ 
  assumes tps' = tps[j1 := nlltape (nss1 @ nss2)]
  shows transforms (tm-extendl j1 j2) tps ttt tps'
   $\langle$ proof $\rangle$ 

The next Turing machine erases the appended list.

definition tm-extendl-erase :: tapeidx  $\Rightarrow$  tapeidx  $\Rightarrow$  machine where
  tm-extendl-erase j1 j2  $\equiv$  tm-extendl j1 j2 ;; tm-erase-cr j2

lemma tm-extendl-erase-tm:
  assumes  $0 < j1$  and  $0 < j2$  and  $G \geq 4$  and  $j1 < k$  and  $j2 < k$ 
  shows turing-machine k G (tm-extendl-erase j1 j2)
   $\langle$ proof $\rangle$ 

lemma transforms-tm-extendl-eraseI [transforms-intros]:
  fixes tps tps' :: tape list and j1 j2 :: tapeidx and ttt k :: nat and nss1 nss2 :: nat list list
  assumes  $0 < j1$   $j1 < k$   $j2 < k$   $j1 \neq j2$   $0 < j2$  length tps = k
  assumes

```

```

    tps ! j1 = nlltape nss1
    tps ! j2 = (|nss2|NLL, 1)
assumes ttt = 11 + 4 * nlllength nss2
assumes tps' = tps
    [j1 := nlltape (nss1 @ nss2),
     j2 := (|[]|, 1)]
shows transforms (tm-extendl-erase j1 j2) tps ttt tps'
⟨proof⟩

```

## 2.9.4 Moving to the next element

Iterating over a list of lists of numbers works with the same Turing machine, *tm-nextract*, as for lists of numbers. We just have to set the parameter  $z$  to the terminator symbol  $\sharp$ . For the proof we can also just copy from the previous section, replacing the symbol  $|$  by  $\sharp$  and *nllength* by *nlllength*, etc.

```

locale turing-machine-nextract-5 =
  fixes j1 j2 :: tapeidx
begin

```

```

definition tm1 ≡ tm-erase-cr j2
definition tm2 ≡ tm1 ;; tm-cp-until j1 j2 { $\sharp$ }
definition tm3 ≡ tm2 ;; tm-cr j2
definition tm4 ≡ tm3 ;; tm-right j1

```

```

lemma tm4-eq-tm-nextract: tm4 = tm-nextract  $\sharp$  j1 j2
⟨proof⟩

```

**context**

```

fixes tps0 :: tape list and k idx :: nat and nss :: nat list list and dummy :: nat list
assumes jk: j1 < k j2 < k 0 < j1 0 < j2 j1 ≠ j2 length tps0 = k
and idx: idx < length nss
and tps0:
    tps0 ! j1 = nlltape' nss idx
    tps0 ! j2 = (|dummy|NL, 1)

```

**begin**

```

definition tps1 ≡ tps0[j2 := (|[]|NL, 1)]

```

```

lemma tm1 [transforms-intros]:
assumes ttt = 7 + 2 * nlllength dummy
shows transforms tm1 tps0 ttt tps1
⟨proof⟩

```

```

definition tps2 ≡ tps0
    [j1 := (|nss|NLL, nlllength (take (Suc idx) nss)),
     j2 := (|nss ! idx|NL, Suc (nlllength (nss ! idx)))]

```

```

lemma tm2 [transforms-intros]:
assumes ttt = 7 + 2 * nlllength dummy + Suc (nlllength (nss ! idx))
shows transforms tm2 tps0 ttt tps2
⟨proof⟩

```

```

definition tps3 ≡ tps0
    [j1 := (|nss|NLL, nlllength (take (Suc idx) nss)),
     j2 := (|nss ! idx|NL, 1)]

```

```

lemma tm3 [transforms-intros]:
assumes ttt = 11 + 2 * nlllength dummy + 2 * (nlllength (nss ! idx))
shows transforms tm3 tps0 ttt tps3
⟨proof⟩

```

```

definition tps4 ≡ tps0
    [j1 := nlltape' nss (Suc idx),

```

$j2 := (\lfloor nss \uparrow idx \rfloor_{NL}, 1)$

**lemma** *tm4*:  
**assumes**  $ttt = 12 + 2 * nlength\ dummy + 2 * (nlength\ (nss \uparrow idx))$   
**shows** *transforms tm4 tps0 ttt tps4*  
*<proof>*

**end**

**end**

**lemma** *transforms-tm-nextract-5I* [*transforms-intros*]:  
**fixes**  $j1\ j2 :: tapeidx$   
**fixes**  $tps\ tps' :: tape\ list$  **and**  $k\ idx :: nat$  **and**  $nss :: nat\ list\ list$  **and**  $dummy :: nat\ list$   
**assumes**  $j1 < k\ j2 < k\ 0 < j1\ 0 < j2\ j1 \neq j2\ length\ tps = k$   
**and**  $idx < length\ nss$   
**assumes**  
 $tps \uparrow j1 = nlltape' nss\ idx$   
 $tps \uparrow j2 = (\lfloor dummy \rfloor_{NL}, 1)$   
**assumes**  $ttt = 12 + 2 * nlength\ dummy + 2 * (nlength\ (nss \uparrow idx))$   
**assumes**  $tps' = tps$   
 $[j1 := nlltape' nss\ (Suc\ idx),$   
 $j2 := (\lfloor nss \uparrow idx \rfloor_{NL}, 1)]$   
**shows** *transforms (tm-nextract # j1 j2) tps ttt tps'*  
*<proof>*

**end**

## 2.10 Mapping between a binary and a quaternary alphabet

**theory** *Two-Four-Symbols*  
**imports** *Arithmetic*  
**begin**

Functions are defined over bit strings. For Turing machines these bits are represented by the symbols **0** and **1**. Sometimes we want a TM to receive a pair of bit strings or output a list of numbers. Or we might want the TM to interpret the input as a list of lists of numbers. All these objects can naturally be represented over a four-symbol (quaternary) alphabet, as we have seen for pairs in Section 2.1.3 and for the lists in Sections 2.8 and 2.9.

To accommodate the aforementioned use cases, we define a straightforward mapping between the binary alphabet **01** and the quaternary alphabet **01#** and devise Turing machines to encode and decode symbol sequences.

### 2.10.1 Encoding and decoding

The encoding maps:

**0**  $\mapsto$  **00**  
**1**  $\mapsto$  **01**  
 $|$   $\mapsto$  **10**  
 $\#$   $\mapsto$  **11**

For example, the list  $[6, 0, 1]$  is represented by the symbols **011|1**, which is encoded as **00010110100110**. Pairing this sequence with the symbol sequence **0110** yields **00010110100110#0110**, which is encoded as **00000001000101000100000101001100010100**.

**definition** *binencode*  $:: symbol\ list \Rightarrow symbol\ list$  **where**  
 $binencode\ ys \equiv concat\ (map\ (\lambda y. [tosym\ ((y - 2)\ div\ 2),\ tosym\ ((y - 2)\ mod\ 2)]))\ ys)$

**lemma** *length-binencode* [*simp*]:  $length\ (binencode\ ys) = 2 * length\ ys$   
*<proof>*

**lemma** *binencode-snoc*:  
 $\text{binencode } (zs \text{ @ } [\mathbf{0}]) = \text{binencode } zs \text{ @ } [\mathbf{0}, \mathbf{0}]$   
 $\text{binencode } (zs \text{ @ } [\mathbf{1}]) = \text{binencode } zs \text{ @ } [\mathbf{0}, \mathbf{1}]$   
 $\text{binencode } (zs \text{ @ } []) = \text{binencode } zs \text{ @ } [\mathbf{1}, \mathbf{0}]$   
 $\text{binencode } (zs \text{ @ } [\#]) = \text{binencode } zs \text{ @ } [\mathbf{1}, \mathbf{1}]$   
 $\langle \text{proof} \rangle$

**lemma** *binencode-at-even*:  
**assumes**  $i < \text{length } ys$   
**shows**  $\text{binencode } ys ! (2 * i) = 2 + (ys ! i - 2) \text{ div } 2$   
 $\langle \text{proof} \rangle$

**lemma** *binencode-at-odd*:  
**assumes**  $i < \text{length } ys$   
**shows**  $\text{binencode } ys ! (2 * i + 1) = 2 + (ys ! i - 2) \text{ mod } 2$   
 $\langle \text{proof} \rangle$

While *binencode* is defined for arbitrary symbol sequences, we only consider sequences over  $\mathbf{01}\#$  to be binencodable.

**abbreviation** *binencodable* :: *symbol list*  $\Rightarrow$  *bool* **where**  
 $\text{binencodable } ys \equiv \forall i < \text{length } ys. 2 \leq ys ! i \wedge ys ! i < 6$

**lemma** *binencodable-append*:  
**assumes** *binencodable* *xs* **and** *binencodable* *ys*  
**shows** *binencodable* (*xs* @ *ys*)  
 $\langle \text{proof} \rangle$

**lemma** *bit-symbols-binencode*:  
**assumes** *binencodable* *ys*  
**shows** *bit-symbols* (*binencode* *ys*)  
 $\langle \text{proof} \rangle$

An encoded symbol sequence is of even length. When decoding a symbol sequence of odd length, we ignore the last symbol. For example,  $\mathbf{011100}$  and  $\mathbf{0111001}$  are both decoded to  $\mathbf{1}\#0$ .

The bit symbol sequence  $[a, b]$  is decoded to this symbol:

**abbreviation** *decsym* :: *symbol*  $\Rightarrow$  *symbol*  $\Rightarrow$  *symbol* **where**  
 $\text{decsym } a \ b \equiv \text{tosym } (2 * \text{todigit } a + \text{todigit } b)$

**definition** *bindecode* :: *symbol list*  $\Rightarrow$  *symbol list* **where**  
 $\text{bindecode } zs \equiv \text{map } (\lambda i. \text{decsym } (zs ! (2 * i)) (zs ! (\text{Suc } (2 * i)))) [0..<\text{length } zs \text{ div } 2]$

**lemma** *length-bindecode* [*simp*]:  $\text{length } (\text{bindecode } zs) = \text{length } zs \text{ div } 2$   
 $\langle \text{proof} \rangle$

**lemma** *bindecode-at*:  
**assumes**  $i < \text{length } zs \text{ div } 2$   
**shows**  $\text{bindecode } zs ! i = \text{decsym } (zs ! (2 * i)) (zs ! (\text{Suc } (2 * i)))$   
 $\langle \text{proof} \rangle$

**lemma** *proper-bindecode*: *proper-symbols* (*bindecode* *zs*)  
 $\langle \text{proof} \rangle$

**lemma** *bindecode2345*:  
**assumes** *bit-symbols* *zs*  
**shows**  $\forall i < \text{length } (\text{bindecode } zs). \text{bindecode } zs ! i \in \{2..<6\}$   
 $\langle \text{proof} \rangle$

**lemma** *bindecode-odd*:  
**assumes**  $\text{length } zs = 2 * n + 1$   
**shows**  $\text{bindecode } zs = \text{bindecode } (\text{take } (2 * n) \ zs)$   
 $\langle \text{proof} \rangle$

**lemma** *bindecode-append*:

```

assumes even (length ys) and even (length zs)
shows bindecode (ys @ zs) = bindecode ys @ bindecode zs
  (is ?lhs = ?rhs)
⟨proof⟩

```

```

lemma bindecode-take-snoc:
  assumes i < length zs div 2
  shows bindecode (take (2 * i) zs) @ [decsym (zs ! (2*i)) (zs ! (Suc (2*i)))] = bindecode (take (2 * Suc i) zs)
⟨proof⟩

```

```

lemma bindecode-encode:
  assumes binencodable ys
  shows bindecode (binencode ys) = ys
⟨proof⟩

```

```

lemma binencode-decode:
  assumes bit-symbols zs and even (length zs)
  shows binencode (bindecode zs) = zs
⟨proof⟩

```

```

lemma binencode-inj:
  assumes binencodable xs and binencodable ys and binencode xs = binencode ys
  shows xs = ys
⟨proof⟩

```

## 2.10.2 Turing machines for encoding and decoding

The next Turing machine implements *binencode* for *binencodable* symbol sequences. It expects a symbol sequence *zs* on tape  $j_1$  and writes *binencode zs* to tape  $j_2$ .

**definition** *tm-binencode* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-binencode j1 j2  $\equiv$ 
  WHILE [] ;  $\lambda$ rs. rs ! j1  $\neq$   $\square$  DO
    IF  $\lambda$ rs. rs ! j1 = 0 THEN
      tm-print j2 [0, 0]
    ELSE
      IF  $\lambda$ rs. rs ! j1 = 1 THEN
        tm-print j2 [0, 1]
      ELSE
        IF  $\lambda$ rs. rs ! j1 = | THEN
          tm-print j2 [1, 0]
        ELSE
          tm-print j2 [1, 1]
        ENDIF
      ENDIF
    ENDIF ;;
  tm-right j1
  DONE

```

```

lemma tm-binencode-tm:
  assumes k  $\geq$  2 and G  $\geq$  4 and j1 < k and j2 < k and 0 < j2
  shows turing-machine k G (tm-binencode j1 j2)
⟨proof⟩

```

```

locale turing-machine-binencode =
  fixes j1 j2 :: tapeidx
begin

```

```

definition tm1  $\equiv$  IF  $\lambda$ rs. rs ! j1 = | THEN tm-print j2 [1, 0] ELSE tm-print j2 [1, 1] ENDIF

```

```

definition tm2  $\equiv$  IF  $\lambda$ rs. rs ! j1 = 1 THEN tm-print j2 [0, 1] ELSE tm1 ENDIF

```

```

definition tm3  $\equiv$  IF  $\lambda$ rs. rs ! j1 = 0 THEN tm-print j2 [0, 0] ELSE tm2 ENDIF

```

```

definition tm4  $\equiv$  tm3 ;; tm-right j1

```

```

definition tm5  $\equiv$  WHILE [] ;  $\lambda$ rs. rs ! j1  $\neq$   $\square$  DO tm4 DONE

```

**lemma** *tm5-eq-tm-binencode*:  $tm5 = tm\text{-}binencode\ j1\ j2$   
 ⟨*proof*⟩

**context**

**fixes**  $k :: nat$  **and**  $tps0 :: tape\ list$  **and**  $zs :: symbol\ list$   
**assumes**  $jk$ :  $k = length\ tps0\ j1 \neq j2\ 0 < j2\ j1 < k\ j2 < k$   
**assumes**  $zs$ : *binencodable*  $zs$   
**assumes**  $tps0$ :  
 $tps0 ! j1 = (\lfloor zs \rfloor, 1)$   
 $tps0 ! j2 = (\lfloor [] \rfloor, 1)$

**begin**

**definition**  $tpsL :: nat \Rightarrow tape\ list$  **where**

$tpsL\ t \equiv tps0$   
 $[j1 := (\lfloor zs \rfloor, Suc\ t),$   
 $j2 := (\lfloor binencode\ (take\ t\ zs) \rfloor, Suc\ (2 * t))]$

**lemma** *tpsL-0*:  $tpsL\ 0 = tps0$   
 ⟨*proof*⟩

**definition**  $tpsL1 :: nat \Rightarrow tape\ list$  **where**

$tpsL1\ t \equiv tps0$   
 $[j1 := (\lfloor zs \rfloor, Suc\ t),$   
 $j2 := (\lfloor binencode\ (take\ (Suc\ t)\ zs) \rfloor, Suc\ (2 * t) + 2)]$

**lemma** *read-tpsL*:

**assumes**  $t < length\ zs$   
**shows**  $read\ (tpsL\ t) ! j1 = zs ! t$

⟨*proof*⟩

**lemma** *tm1* [*transforms-intros*]:

**assumes**  $t < length\ zs$  **and**  $zs ! t = | \vee zs ! t = \#$   
**shows**  $transforms\ tm1\ (tpsL\ t)\ 4\ (tpsL1\ t)$   
 ⟨*proof*⟩

**lemma** *tm2* [*transforms-intros*]:

**assumes**  $t < length\ zs$  **and**  $zs ! t = | \vee zs ! t = \# \vee zs ! t = \mathbf{1}$   
**shows**  $transforms\ tm2\ (tpsL\ t)\ 5\ (tpsL1\ t)$   
 ⟨*proof*⟩

**lemma** *tm3* [*transforms-intros*]:

**assumes**  $t < length\ zs$   
**shows**  $transforms\ tm3\ (tpsL\ t)\ 6\ (tpsL1\ t)$   
 ⟨*proof*⟩

**lemma** *tm4* [*transforms-intros*]:

**assumes**  $t < length\ zs$   
**shows**  $transforms\ tm4\ (tpsL\ t)\ 7\ (tpsL\ (Suc\ t))$   
 ⟨*proof*⟩

**lemma** *tm5*:

**assumes**  $t = 9 * length\ zs + 1$   
**shows**  $transforms\ tm5\ (tpsL\ 0)\ t\ (tpsL\ (length\ zs))$   
 ⟨*proof*⟩

**lemma** *tpsL*:  $tpsL\ (length\ zs) = tps0$

$[j1 := (\lfloor zs \rfloor, Suc\ (length\ zs)),$   
 $j2 := (\lfloor binencode\ zs \rfloor, Suc\ (2 * (length\ zs)))]$   
 ⟨*proof*⟩

**lemma** *tm5'*:

**assumes**  $t = 9 * length\ zs + 1$   
**shows**  $transforms\ tm5\ tps0\ t\ (tpsL\ (length\ zs))$

*<proof>*

**end**

**end**

**lemma** *transforms-tm-binencodeI* [*transforms-intros*]:

**fixes** *j1 j2* :: *tapeidx*

**fixes** *tps tps'* :: *tape list* **and** *ttt k* :: *nat* **and** *zs* :: *symbol list*

**assumes** *k = length tps j1 ≠ j2 0 < j2 j1 < k j2 < k*

**and** *binencodable zs*

**assumes**

*tps ! j1 = (⌊zs⌋, 1)*

*tps ! j2 = (⌊[]⌋, 1)*

**assumes** *ttt = 9 \* length zs + 1*

**assumes** *tps' ≡ tps*

*[j1 := (⌊zs⌋, Suc (length zs)),*

*j2 := (⌊binencode zs⌋, Suc (2 \* (length zs)))]*

**shows** *transforms (tm-binencode j1 j2) tps ttt tps'*

*<proof>*

The next command reads chunks of two symbols over **01** from one tape and writes to another tape the corresponding symbol over **01**‡. The first symbol of each chunk is memorized on the last tape. If the end of the input (that is, a blank symbol) is encountered, the command stops without writing another symbol.

**definition** *cmd-bindec* :: *tapeidx* ⇒ *tapeidx* ⇒ *command* **where**

*cmd-bindec j1 j2 rs* ≡

*if rs ! j1 = 0 then (1, map (λz. (z, Stay)) rs)*

*else (0,*

*map (λi.*

*if last rs = ▷*

*then if i = j1 then (rs ! i, Right)*

*else if i = j2 then (rs ! i, Stay)*

*else if i = length rs - 1 then (tosym (todigit (rs ! j1)), Stay)*

*else (rs ! i, Stay)*

*else if i = j1 then (rs ! i, Right)*

*else if i = j2 then (decsym (last rs) (rs ! j1), Right)*

*else if i = length rs - 1 then (1, Stay)*

*else (rs ! i, Stay))*

*[0..*length rs*]*

The Turing machine performing the decoding:

**definition** *tm-bindec* :: *tapeidx* ⇒ *tapeidx* ⇒ *machine* **where**

*tm-bindec j1 j2 = [cmd-bindec j1 j2]*

**context**

**fixes** *j1 j2* :: *tapeidx* **and** *k* :: *nat*

**assumes** *j-less: j1 < k j2 < k*

**and** *j-gt: 0 < j2*

**begin**

**lemma** *turing-command-bindec*:

**assumes** *G ≥ 6*

**shows** *turing-command (Suc k) 1 G (cmd-bindec j1 j2)*

*<proof>*

**lemma** *tm-bindec-tm*: *G ≥ 6 ⇒ turing-machine (Suc k) G (tm-bindec j1 j2)*

*<proof>*

**context**

**fixes** *tps* :: *tape list* **and** *zs* :: *symbol list*

**assumes** *j1-neq: j1 ≠ j2*

**and** *lentps: Suc k = length tps*

```

    and bs: bit-symbols zs
begin

lemma sem-cmd-bindec-gt:
  assumes tps ! j1 = ( $\lfloor$ zs $\rfloor$ , i)
    and i > length zs
  shows sem (cmd-bindec j1 j2) (0, tps) = (1, tps)
<proof>

lemma sem-cmd-bindec-1:
  assumes tps ! k =  $\lceil$  $\triangleright$ 
    and tps ! j1 = ( $\lfloor$ zs $\rfloor$ , i)
    and i > 0
    and i  $\leq$  length zs
    and tps' = tps [j1 := tps ! j1 |+| 1, k :=  $\lceil$ todigit (tps :: j1) + 2 $\rceil$ ]
  shows sem (cmd-bindec j1 j2) (0, tps) = (0, tps')
<proof>

lemma sem-cmd-bindec-23:
  assumes tps ! k =  $\lceil$ s $\rceil$ 
    and s = 0  $\vee$  s = 1
    and tps ! j1 = ( $\lfloor$ zs $\rfloor$ , i)
    and i > 0
    and i  $\leq$  length zs
    and tps' = tps
      [j1 := tps ! j1 |+| 1,
       j2 := tps ! j2 |:=| decsym s (tps :: j1) |+| 1,
       k :=  $\lceil$  $\triangleright$  $\rceil$ ]
  shows sem (cmd-bindec j1 j2) (0, tps) = (0, tps')
<proof>

end

lemma transits-tm-bindec:
  fixes tps :: tape list and zs :: symbol list
  assumes j1-neq: j1  $\neq$  j2
    and j1j2: 0 < j2 j1 < k j2 < k
    and lentps: Suc k = length tps
    and bs: bit-symbols zs
  assumes tps ! k =  $\lceil$  $\triangleright$ 
    and tps ! j1 = ( $\lfloor$ zs $\rfloor$ , 2 * i + 1)
    and tps ! j2 = ( $\lfloor$ bindecode (take (2 * i) zs) $\rfloor$ , Suc i)
    and i < length zs div 2
    and tps' = tps
      [j1 := ( $\lfloor$ zs $\rfloor$ , 2 * (Suc i) + 1),
       j2 := ( $\lfloor$ bindecode (take (2 * Suc i) zs) $\rfloor$ , Suc (Suc i))]
  shows transits (tm-bindec j1 j2) (0, tps) 2 (0, tps')
<proof>

context
  fixes tps :: tape list and zs :: symbol list
  assumes j1-neq: j1  $\neq$  j2
    and j1j2: 0 < j2 j1 < k j2 < k
    and lentps: Suc k = length tps
    and bs: bit-symbols zs
begin

lemma transits-tm-bindec':
  assumes tps ! k =  $\lceil$  $\triangleright$ 
    and tps ! j1 = ( $\lfloor$ zs $\rfloor$ , 1)
    and tps ! j2 = ( $\lfloor$  $\square$  $\rfloor$ , 1)
    and i  $\leq$  length zs div 2
    and tps' = tps

```

```

    [j1 := ([zs], 2 * i + 1),
     j2 := ([bindecode (take (2 * i) zs)], Suc i)]
  shows transits (tm-bindec j1 j2) (0, tps) (2 * i) (0, tps')
  ⟨proof⟩

```

**corollary** *transits-tm-bindec''*:

```

  assumes tps ! k = [▷]
  and tps ! j1 = ([zs], 1)
  and tps ! j2 = ([[]], 1)
  and l = length zs div 2
  and tps' = tps
  [j1 := ([zs], 2 * l + 1),
   j2 := ([bindecode (take (2 * l) zs)], Suc l)]
  shows transits (tm-bindec j1 j2) (0, tps) (2 * l) (0, tps')
  ⟨proof⟩

```

In case the input is of odd length, that is, malformed:

**lemma** *transforms-tm-bindec-odd*:

```

  assumes tps ! k = [▷]
  and tps ! j1 = ([zs], 1)
  and tps ! j2 = ([[]], 1)
  and tps' = tps
  [j1 := ([zs], 2 * l + 2),
   j2 := ([bindecode zs], Suc l),
   k := [todigit (last zs) + 2]]
  and l = length zs div 2
  and Suc (2 * l) = length zs
  shows transforms (tm-bindec j1 j2) tps (2 * l + 2) tps'
  ⟨proof⟩

```

In case the input is of even length, that is, properly encoded:

**lemma** *transforms-tm-bindec-even*:

```

  assumes tps ! k = [▷]
  and tps ! j1 = ([zs], 1)
  and tps ! j2 = ([[]], 1)
  and tps' = tps
  [j1 := ([zs], 2 * l + 1),
   j2 := ([bindecode zs], Suc l)]
  and l = length zs div 2
  and 2 * l = length zs
  shows transforms (tm-bindec j1 j2) tps (2 * l + 1) tps'
  ⟨proof⟩

```

**lemma** *transforms-tm-bindec*:

```

  assumes tps ! k = [▷]
  and tps ! j1 = ([zs], 1)
  and tps ! j2 = ([[]], 1)
  and tps' = tps
  [j1 := ([zs], Suc (length zs)),
   j2 := ([bindecode zs], Suc (length zs div 2)),
   k := [if even (length zs) then 1 else (todigit (last zs) + 2)]]
  shows transforms (tm-bindec j1 j2) tps (Suc (length zs)) tps'
  ⟨proof⟩

```

**end**

**end**

Next we eliminate the memorization tape from *tm-bindec*.

**lemma** *transforms-cartesian-bindec*:

```

  assumes G ≥ (6 :: nat)
  assumes j1 ≠ j2
  and j1j2: 0 < j2 j1 < k j2 < k

```

```

and  $k = \text{length } tps$ 
and  $\text{bit-symbols } zs$ 
assumes  $tps ! j1 = (\lfloor zs \rfloor, 1)$ 
and  $tps ! j2 = (\lfloor [] \rfloor, 1)$ 
assumes  $t = \text{Suc } (\text{length } zs)$ 
and  $tps' = tps$ 
 $[j1 := (\lfloor zs \rfloor, \text{Suc } (\text{length } zs)),$ 
 $j2 := (\lfloor \text{bindecode } zs \rfloor, \text{Suc } (\text{length } zs \text{ div } 2))]$ 
shows  $\text{transforms } (\text{cartesian } (\text{tm-bindec } j1 j2) 4) tps t tps'$ 
<proof>

```

The next Turing machine decodes a bit symbol sequence given on tape  $j_1$  into a quaternary symbol sequence output to tape  $j_2$ . It executes the previous TM followed by carriage returns on the tapes  $j_1$  and  $j_2$ .

**definition**  $\text{tm-bindecode} :: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**  
 $\text{tm-bindecode } j1 j2 \equiv \text{cartesian } (\text{tm-bindec } j1 j2) 4 ;; \text{tm-cr } j1 ;; \text{tm-cr } j2$

**lemma**  $\text{tm-bindecode-tm}$ :  
**fixes**  $j1 j2 :: \text{tapeidx}$  **and**  $G k :: \text{nat}$   
**assumes**  $G \geq 6$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $0 < j2$  **and**  $j1 \neq j2$   
**shows**  $\text{turing-machine } k G (\text{tm-bindecode } j1 j2)$   
<proof>

**locale**  $\text{turing-machine-bindecode} =$   
**fixes**  $j1 j2 :: \text{tapeidx}$   
**begin**

**definition**  $\text{tm1} \equiv \text{cartesian } (\text{tm-bindec } j1 j2) 4$   
**definition**  $\text{tm2} \equiv \text{tm1} ;; \text{tm-cr } j1$   
**definition**  $\text{tm3} \equiv \text{tm2} ;; \text{tm-cr } j2$

**lemma**  $\text{tm3-eq-tm-bindecode}$ :  $\text{tm3} = \text{tm-bindecode } j1 j2$   
<proof>

**context**  
**fixes**  $tps0 :: \text{tape list}$  **and**  $zs :: \text{symbol list}$  **and**  $k :: \text{nat}$   
**assumes**  $jk: j1 < k j2 < k 0 < j2 j1 \neq j2 k = \text{length } tps0$   
**assumes**  $zs: \text{bit-symbols } zs$   
**assumes**  $tps0$ :  
 $tps0 ! j1 = (\lfloor zs \rfloor, 1)$   
 $tps0 ! j2 = (\lfloor [] \rfloor, 1)$   
**begin**

**definition**  $\text{tps1} \equiv tps0$   
 $[j1 := (\lfloor zs \rfloor, \text{Suc } (\text{length } zs)),$   
 $j2 := (\lfloor \text{bindecode } zs \rfloor, \text{Suc } (\text{length } zs \text{ div } 2))]$

**lemma**  $\text{tm1}$  [ $\text{transforms-intros}$ ]:  
**assumes**  $t = \text{Suc } (\text{length } zs)$   
**shows**  $\text{transforms } \text{tm1 } tps0 t tps1$   
<proof>

**definition**  $\text{tps2} \equiv tps0$   
 $[j2 := (\lfloor \text{bindecode } zs \rfloor, \text{Suc } (\text{length } zs \text{ div } 2))]$

**lemma**  $\text{tm2}$  [ $\text{transforms-intros}$ ]:  
**assumes**  $t = 2 * \text{length } zs + 4$   
**shows**  $\text{transforms } \text{tm2 } tps0 t tps2$   
<proof>

**definition**  $\text{tps3} \equiv tps0$   
 $[j2 := (\lfloor \text{bindecode } zs \rfloor, 1)]$

```

lemma tm3:
  assumes  $t = 2 * \text{length } zs + 7 + \text{length } zs \text{ div } 2$ 
  shows transforms tm3 tps0 t tps3
  <proof>

```

```

lemma tm3':
  assumes  $t = 7 + 3 * \text{length } zs$ 
  shows transforms tm3 tps0 t tps3
  <proof>

```

**end**

**end**

```

lemma transforms-tm-bindecodeI [transforms-intros]:
  fixes  $j1\ j2 :: \text{tapeidx}$ 
  fixes  $tps :: \text{tape list}$  and  $zs :: \text{symbol list}$  and  $k\ ttt :: \text{nat}$ 
  assumes  $j1 < k$  and  $j2 < k$  and  $0 < j2$  and  $j1 \neq j2$  and  $k = \text{length } tps$ 
  and bit-symbols zs
  assumes
     $tps ! j1 = (\lfloor zs \rfloor, 1)$ 
     $tps ! j2 = (\lfloor \square \rfloor, 1)$ 
  assumes  $ttt = 7 + 3 * \text{length } zs$ 
  assumes  $tps' = tps$ 
     $[j2 := (\lfloor \text{bindecode } zs \rfloor, 1)]$ 
  shows transforms (tm-bindecode j1 j2) tps ttt tps'
  <proof>

```

**end**

## 2.11 Symbol sequence operations

```

theory Symbol-Ops
  imports Two-Four-Symbols
begin

```

While previous sections have focused on “formatted” symbol sequences for numbers and lists, in this section we devise some Turing machines dealing with “unstructured” arbitrary symbol sequences. The only “structure” that is often imposed is that of not containing a blank symbol because when reading a symbol sequence, say from the input tape, a blank would signal the end of the symbol sequence.

### 2.11.1 Checking for being over an alphabet

In this section we devise a Turing machine that checks if a proper symbol sequence is over a given alphabet represented by an upper bound symbol  $z$ .

```

abbreviation proper-symbols-lt :: symbol  $\Rightarrow$  symbol list  $\Rightarrow$  bool where
  proper-symbols-lt  $z\ zs \equiv \text{proper-symbols } zs \wedge \text{symbols-lt } z\ zs$ 

```

The next Turing machine checks if the symbol sequence (up until the first blank) on tape  $j_1$  contains only symbols from  $\{2, \dots, z-1\}$ , where  $z$  is a parameter of the TM, and writes to tape  $j_2$  the number 1 or 0, representing True or False, respectively. It assumes that  $j_2$  initially contains at most one symbol.

```

definition tm-proper-symbols-lt :: tapeidx  $\Rightarrow$  tapeidx  $\Rightarrow$  symbol  $\Rightarrow$  machine where
  tm-proper-symbols-lt  $j1\ j2\ z \equiv$ 
    tm-write  $j2\ \mathbf{1}$  ;;
    WHILE  $\square$  ;  $\lambda rs. rs ! j1 \neq \square$  DO
      IF  $\lambda rs. rs ! j1 < 2 \vee rs ! j1 \geq z$  THEN
        tm-write  $j2\ \square$ 
      ELSE
         $\square$ 
      ENDIF ;;
    tm-right  $j1$ 

```

*DONE* ;;  
*tm-cr j1*

**lemma** *tm-proper-symbols-lt-tm*:  
**assumes**  $0 < j2$   $j1 < k$   $j2 < k$  **and**  $G \geq 4$   
**shows** *turing-machine k G (tm-proper-symbols-lt j1 j2 z)*  
*<proof>*

**locale** *turing-machine-proper-symbols-lt* =  
**fixes**  $j1$   $j2$  :: *tapeidx* **and**  $z$  :: *symbol*  
**begin**

**definition**  $tm1 \equiv tm\text{-write } j2$  **1**  
**definition**  $tm2 \equiv IF \lambda rs. rs ! j1 < 2 \vee rs ! j1 \geq z THEN tm\text{-write } j2 \square ELSE \square$  *ENDIF*  
**definition**  $tm3 \equiv tm2$  ;; *tm-right j1*  
**definition**  $tm4 \equiv WHILE \square ; \lambda rs. rs ! j1 \neq \square DO tm3$  *DONE*  
**definition**  $tm5 \equiv tm1$  ;;  $tm4$   
**definition**  $tm6 \equiv tm5$  ;; *tm-cr j1*

**lemma** *tm6-eq-tm-proper-symbols-lt*:  $tm6 = tm\text{-proper-symbols-lt } j1$   $j2$   $z$   
*<proof>*

**context**  
**fixes**  $zs$  :: *symbol list* **and**  $tps0$  :: *tape list* **and**  $k$  :: *nat*  
**assumes**  $jk$ :  $k = length$   $tps0$   $j1 \neq j2$   $j1 < k$   $j2 < k$   
**and**  $zs$ : *proper-symbols zs*  
**and**  $tps0$ :  
 $tps0 ! j1 = (\lfloor zs \rfloor, 1)$   
 $tps0 ! j2 = (\lfloor \square \rfloor, 1)$

**begin**

**definition**  $tps1$   $t \equiv tps0$   
 $j1 := (\lfloor zs \rfloor, Suc\ t),$   
 $j2 := (if\ proper\text{-symbols}\text{-lt}\ z\ (take\ t\ zs)\ then\ \lfloor \mathbf{1} \rfloor\ else\ \lfloor \square \rfloor, 1)$

**lemma**  $tm1$  [*transforms-intros*]: *transforms tm1 tps0 1 (tps1 0)*  
*<proof>*

**definition**  $tps2$   $t \equiv tps0$   
 $j1 := (\lfloor zs \rfloor, Suc\ t),$   
 $j2 := (if\ proper\text{-symbols}\text{-lt}\ z\ (take\ (Suc\ t)\ zs)\ then\ \lfloor \mathbf{1} \rfloor\ else\ \lfloor \square \rfloor, 1)$

**lemma**  $tm2$  [*transforms-intros*]:  
**assumes**  $t < length$   $zs$   
**shows** *transforms tm2 (tps1 t) 3 (tps2 t)*  
*<proof>*

**lemma**  $tm3$  [*transforms-intros*]:  
**assumes**  $t < length$   $zs$   
**shows** *transforms tm3 (tps1 t) 4 (tps1 (Suc t))*  
*<proof>*

**lemma**  $tm4$  [*transforms-intros*]:  
**assumes**  $t = 1 + 6 * length$   $zs$   
**shows** *transforms tm4 (tps1 0) t (tps1 (length zs))*  
*<proof>*

**lemma**  $tm5$  [*transforms-intros*]:  
**assumes**  $t = 2 + 6 * length$   $zs$   
**shows** *transforms tm5 tps0 t (tps1 (length zs))*  
*<proof>*

**definition**  $tps5 \equiv tps0$

```

[j1 := ( $\lfloor zs \rfloor$ , 1),
 j2 := (if proper-symbols-lt z zs then  $\lfloor [1] \rfloor$  else  $\lfloor [] \rfloor$ , 1)]

```

**definition**  $tps5' \equiv tps0$

```

[j2 := (if proper-symbols-lt z zs then  $\lfloor [1] \rfloor$  else  $\lfloor [] \rfloor$ , 1)]

```

**lemma**  $tm6$ :

```

assumes  $ttt = 5 + 7 * \text{length } zs$ 
shows transforms  $tm6$   $tps0$   $ttt$   $tps5'$ 
<proof>

```

**definition**  $tps6 \equiv tps0$

```

[j2 := ( $\lfloor \text{proper-symbols-lt } z \text{ } zs \rfloor_B$ , 1)]

```

**lemma**  $tm6'$ :

```

assumes  $ttt = 5 + 7 * \text{length } zs$ 
shows transforms  $tm6$   $tps0$   $ttt$   $tps6$ 
<proof>

```

**end**

**end**

**lemma** *transforms-tm-proper-symbols-ltI* [*transforms-intros*]:

```

fixes  $j1$   $j2$  :: tapeidx and  $z$  :: symbol
fixes  $zs$  :: symbol list and  $tps$   $tps'$  :: tape list and  $k$  :: nat
assumes  $k = \text{length } tps$   $j1 \neq j2$   $j1 < k$   $j2 < k$ 
and proper-symbols  $zs$ 
assumes
   $tps ! j1 = (\lfloor zs \rfloor, 1)$ 
   $tps ! j2 = (\lfloor [] \rfloor, 1)$ 
assumes  $ttt = 5 + 7 * \text{length } zs$ 
assumes  $tps' = tps$ 
   $[j2 := (\lfloor \text{proper-symbols-lt } z \text{ } zs \rfloor_B, 1)]$ 
shows transforms ( $tm\text{-proper-symbols-lt } j1$   $j2$   $z$ )  $tps$   $ttt$   $tps'$ 
<proof>

```

## 2.11.2 The length of the input

The Turing machine in this section reads the input tape until the first blank and increments a counter on tape  $j$  for every symbol read. In the end it performs a carriage return on the input tape, and tape  $j$  contains the length of the input as binary number. For this to work, tape  $j$  must initially be empty.

**lemma** *proper-tape-read*:

```

assumes proper-symbols  $zs$ 
shows  $|\cdot| (\lfloor zs \rfloor, i) = \square \iff i > \text{length } zs$ 
<proof>

```

**definition**  $tm\text{-length-input} :: \text{tapeidx} \Rightarrow \text{machine}$  **where**

```

 $tm\text{-length-input } j \equiv$ 
  WHILE  $\square$  ;  $\lambda rs. rs ! 0 \neq \square$  DO
     $tm\text{-incr } j$  ;;
     $tm\text{-right } 0$ 
  DONE ;;
   $tm\text{-cr } 0$ 

```

**lemma**  $tm\text{-length-input-tm}$ :

```

assumes  $G \geq 4$  and  $0 < j$  and  $j < k$ 
shows turing-machine  $k$   $G$  ( $tm\text{-length-input } j$ )
<proof>

```

**locale**  $turing\text{-machine-length-input} =$

```

fixes  $j$  :: tapeidx
begin

```

**definition**  $tmL1 \equiv tm-incr\ j$   
**definition**  $tmL2 \equiv tmL1 \ ;\ ;\ tm-right\ 0$   
**definition**  $tm1 \equiv WHILE\ []\ ;\ \lambda rs.\ rs\ !\ 0\ \neq\ \square\ DO\ tmL2\ DONE$   
**definition**  $tm2 \equiv tm1 \ ;\ ;\ tm-cr\ 0$

**lemma**  $tm2-eq-tm-length-input$ :  $tm2 = tm-length-input\ j$   
 $\langle proof \rangle$

**context**

**fixes**  $tps0 :: tape\ list$  **and**  $k :: nat$  **and**  $zs :: symbol\ list$   
**assumes**  $jk$ :  $0 < j < k$   $length\ tps0 = k$   
**and**  $zs$ :  $proper-symbols\ zs$   
**and**  $tps0$ :  
 $tps0\ !\ 0 = (\lfloor zs \rfloor, 1)$   
 $tps0\ !\ j = (\lfloor 0 \rfloor_N, 1)$

**begin**

**definition**  $tpsL :: nat \Rightarrow tape\ list$  **where**  
 $tpsL\ t \equiv tps0[0 := (\lfloor zs \rfloor, 1 + t), j := (\lfloor t \rfloor_N, 1)]$

**lemma**  $tpsL-eq-tps0$ :  $tpsL\ 0 = tps0$   
 $\langle proof \rangle$

**definition**  $tpsL1 :: nat \Rightarrow tape\ list$  **where**  
 $tpsL1\ t \equiv tps0[0 := (\lfloor zs \rfloor, 1 + t), j := (\lfloor Suc\ t \rfloor_N, 1)]$

**definition**  $tpsL2 :: nat \Rightarrow tape\ list$  **where**  
 $tpsL2\ t \equiv tps0[0 := (\lfloor zs \rfloor, 1 + Suc\ t), j := (\lfloor Suc\ t \rfloor_N, 1)]$

**lemma**  $tmL1$  [ $transforms-intros$ ]:  
**assumes**  $t < length\ zs$  **and**  $ttt = 5 + 2 * nlength\ t$   
**shows**  $transforms\ tmL1\ (tpsL\ t)\ ttt\ (tpsL1\ t)$   
 $\langle proof \rangle$

**lemma**  $tmL2$ :  
**assumes**  $t < length\ zs$  **and**  $ttt = 6 + 2 * nlength\ t$   
**shows**  $transforms\ tmL2\ (tpsL\ t)\ ttt\ (tpsL\ (Suc\ t))$   
 $\langle proof \rangle$

**lemma**  $tmL2'$ :  
**assumes**  $t < length\ zs$  **and**  $ttt = 6 + 2 * nlength\ (length\ zs)$   
**shows**  $transforms\ tmL2\ (tpsL\ t)\ ttt\ (tpsL\ (Suc\ t))$   
 $\langle proof \rangle$

**lemma**  $tm1$ :  
**assumes**  $ttt = length\ zs * (8 + 2 * nlength\ (length\ zs)) + 1$   
**shows**  $transforms\ tm1\ (tpsL\ 0)\ ttt\ (tpsL\ (length\ zs))$   
 $\langle proof \rangle$

**lemma**  $tmL'$  [ $transforms-intros$ ]:  
**assumes**  $ttt = 10 * length\ zs^2 + 1$   
**shows**  $transforms\ tm1\ (tpsL\ 0)\ ttt\ (tpsL\ (length\ zs))$   
 $\langle proof \rangle$

**definition**  $tps2 :: tape\ list$  **where**  
 $tps2 \equiv tps0[0 := (\lfloor zs \rfloor, 1), j := (\lfloor length\ zs \rfloor_N, 1)]$

**lemma**  $tm2$ :  
**assumes**  $ttt = 10 * length\ zs^2 + length\ zs + 4$   
**shows**  $transforms\ tm2\ (tpsL\ 0)\ ttt\ tps2$   
 $\langle proof \rangle$

**definition**  $tps2' :: \text{tape list where}$   
 $tps2' \equiv tps0[j := (\lfloor \text{length } zs \rfloor_N, 1)]$

**lemma**  $tm2'$ :  
**assumes**  $ttt = 11 * \text{length } zs \wedge 2 + 4$   
**shows** *transforms*  $tm2$   $tps0$   $ttt$   $tps2'$   
 $\langle \text{proof} \rangle$

**end**

**end**

**lemma** *transforms-tm-length-inputI* [*transforms-intros*]:  
**fixes**  $j :: \text{tapeidx}$   
**fixes**  $tps$   $tps' :: \text{tape list and } k :: \text{nat and } zs :: \text{symbol list}$   
**assumes**  $0 < j < k$   $\text{length } tps = k$   
**and** *proper-symbols*  $zs$   
**assumes**  
 $tps ! 0 = (\lfloor zs \rfloor, 1)$   
 $tps ! j = (\lfloor 0 \rfloor_N, 1)$   
**assumes**  $ttt = 11 * \text{length } zs \wedge 2 + 4$   
**assumes**  $tps' = tps$   
 $[j := (\lfloor \text{length } zs \rfloor_N, 1)]$   
**shows** *transforms* (*tm-length-input*  $j$ )  $tps$   $ttt$   $tps'$   
 $\langle \text{proof} \rangle$

### 2.11.3 Whether the length is even

The next Turing machines reads the symbols on tape  $j_1$  until the first blank and alternates between numbers 0 and 1 on tape  $j_2$ . Then tape  $j_2$  contains the parity of the length of the symbol sequence on tape  $j_1$ . Finally, the TM flips the output once more, so that tape  $j_2$  contains a Boolean indicating whether the length was even or not. We assume tape  $j_2$  is initially empty, that is, represents the number 0.

**definition** *tm-even-length*  $:: \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{machine where}$   
 $tm\text{-}even\text{-}length\ j1\ j2 \equiv$   
 $WHILE\ [] ; \lambda rs. rs ! j1 \neq \square\ DO$   
 $tm\text{-}not\ j2\ ;;$   
 $tm\text{-}right\ j1$   
 $DONE\ ;;$   
 $tm\text{-}not\ j2\ ;;$   
 $tm\text{-}cr\ j1$

**lemma** *tm-even-length-tm*:  
**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $j1 < k$   $0 < j2$   $j2 < k$   
**shows** *turing-machine*  $k$   $G$  (*tm-even-length*  $j1$   $j2$ )  
 $\langle \text{proof} \rangle$

**locale** *turing-machine-even-length* =  
**fixes**  $j1$   $j2 :: \text{tapeidx}$   
**begin**

**definition**  $tmB \equiv tm\text{-}not\ j2\ ;;\ tm\text{-}right\ j1$   
**definition**  $tmL \equiv WHILE\ [] ; \lambda rs. rs ! j1 \neq \square\ DO\ tmB\ DONE$   
**definition**  $tm2 \equiv tmL\ ;;\ tm\text{-}not\ j2$   
**definition**  $tm3 \equiv tm2\ ;;\ tm\text{-}cr\ j1$

**lemma** *tm3-eq-tm-even-length*:  $tm3 = tm\text{-}even\text{-}length\ j1\ j2$   
 $\langle \text{proof} \rangle$

**context**  
**fixes**  $tps0 :: \text{tape list and } k :: \text{nat and } zs :: \text{symbol list}$   
**assumes** *zs*: *proper-symbols*  $zs$   
**assumes** *jk*:  $j1 < k$   $j2 < k$   $j1 \neq j2$   $\text{length } tps0 = k$   
**assumes**  $tps0$ :

```

    tps0 ! j1 = ( $\lfloor$ zs $\rfloor$ , 1)
    tps0 ! j2 = ( $\lfloor$ 0 $\rfloor_N$ , 1)
begin

definition tpsL :: nat  $\Rightarrow$  tape list where
  tpsL t  $\equiv$  tps0
  [j1 := ( $\lfloor$ zs $\rfloor$ , Suc t),
   j2 := ( $\lfloor$ odd t $\rfloor_B$ , 1)]

lemma tpsL0: tpsL 0 = tps0
  <proof>

lemma tmL2 [transforms-intros]: transforms tmB (tpsL t) 4 (tpsL (Suc t))
  <proof>

lemma tmL:
  assumes ttt = 6 * length zs + 1
  shows transforms tmL (tpsL 0) ttt (tpsL (length zs))
  <proof>

lemma tmL' [transforms-intros]:
  assumes ttt = 6 * length zs + 1
  shows transforms tmL tps0 ttt (tpsL (length zs))
  <proof>

definition tps2 :: tape list where
  tps2  $\equiv$  tps0
  [j1 := ( $\lfloor$ zs $\rfloor$ , Suc (length zs)),
   j2 := ( $\lfloor$ even (length zs) $\rfloor_B$ , 1)]

lemma tm2 [transforms-intros]:
  assumes ttt = 6 * length zs + 4
  shows transforms tm2 tps0 ttt tps2
  <proof>

definition tps3 :: tape list where
  tps3  $\equiv$  tps0
  [j1 := ( $\lfloor$ zs $\rfloor$ , 1),
   j2 := ( $\lfloor$ even (length zs) $\rfloor_B$ , 1)]

lemma tm3:
  assumes ttt = 7 * length zs + 7
  shows transforms tm3 tps0 ttt tps3
  <proof>

definition tps3' :: tape list where
  tps3'  $\equiv$  tps0
  [j2 := ( $\lfloor$ even (length zs) $\rfloor_B$ , 1)]

lemma tps3': tps3' = tps3
  <proof>

lemma tm3':
  assumes ttt = 7 * length zs + 7
  shows transforms tm3 tps0 ttt tps3'
  <proof>

end

end

lemma transforms-tm-even-lengthI [transforms-intros]:
  fixes j1 j2 :: tapeidx

```

```

fixes tps tps' :: tape list and k :: nat and zs :: symbol list
assumes j1 < k j2 < k j1 ≠ j2
  and proper-symbols zs
  and length tps = k
assumes
  tps ! j1 = (⌊zs⌋, 1)
  tps ! j2 = (⌊0⌋N, 1)
assumes tps' = tps
  [j2 := (⌊even (length zs)⌋B, 1)]
assumes ttt = 7 * length zs + 7
shows transforms (tm-even-length j1 j2) tps ttt tps'
⟨proof⟩

```

### 2.11.4 Checking for ends-with or empty

The next Turing machine implements a slightly idiosyncratic operation that we use in the next section for checking if a symbol sequence represents a list of numbers. The operation consists in checking if the symbol sequence on tape  $j_1$  either is empty or ends with the symbol  $z$ , which is another parameter of the TM. If the condition is met, the number 1 is written to tape  $j_2$ , otherwise the number 0.

```

definition tm-empty-or-endswith :: tapeidx ⇒ tapeidx ⇒ symbol ⇒ machine where
  tm-empty-or-endswith j1 j2 z ≡
    tm-right-until j1 {□} ;;
    tm-left j1 ;;
    IF λrs. rs ! j1 ∈ {▷, z} THEN
      tm-setn j2 1
    ELSE
      []
    ENDIF ;;
    tm-cr j1

```

```

lemma tm-empty-or-endswith-tm:
assumes k ≥ 2 and G ≥ 4 and 0 < j2 and j1 < k and j2 < k
shows turing-machine k G (tm-empty-or-endswith j1 j2 z)
⟨proof⟩

```

```

locale turing-machine-empty-or-endswith =
  fixes j1 j2 :: tapeidx and z :: symbol
begin

```

```

definition tm1 ≡ tm-right-until j1 {□}
definition tm2 ≡ tm1 ;; tm-left j1
definition tmI ≡ IF λrs. rs ! j1 ∈ {▷, z} THEN tm-setn j2 1 ELSE [] ENDIF
definition tm3 ≡ tm2 ;; tmI
definition tm4 ≡ tm3 ;; tm-cr j1

```

```

lemma tm4-eq-tm-empty-or-endswith: tm4 = tm-empty-or-endswith j1 j2 z
⟨proof⟩

```

```

context
fixes tps0 :: tape list and k :: nat and zs :: symbol list
assumes jk: j1 ≠ j2 j1 < k j2 < k length tps0 = k
  and zs: proper-symbols zs
  and tps0:
    tps0 ! j1 = (⌊zs⌋, 1)
    tps0 ! j2 = (⌊0⌋N, 1)
begin

```

```

definition tps1 :: tape list where
  tps1 ≡ tps0
  [j1 := (⌊zs⌋, Suc (length zs))]

```

```

lemma tm1 [transforms-intros]:
assumes ttt = Suc (length zs)

```

**shows** *transforms tm1 tps0 ttt tps1*  
 ⟨*proof*⟩

**definition** *tps2* :: *tape list* **where**  
*tps2*  $\equiv$  *tps0*  
 [*j1* := ( $\lfloor$ zs $\rfloor$ , *length* zs)]

**lemma** *tm2* [*transforms-intros*]:  
**assumes** *ttt* = 2 + *length* zs  
**shows** *transforms tm2 tps0 ttt tps2*  
 ⟨*proof*⟩

**definition** *tps3* :: *tape list* **where**  
*tps3*  $\equiv$  *tps0*  
 [*j1* := ( $\lfloor$ zs $\rfloor$ , *length* zs),  
*j2* := ( $\lfloor$ zs = []  $\vee$  last zs = z $\rfloor_B$ , 1)]

**lemma** *tmI* [*transforms-intros*]: *transforms tmI tps2 14 tps3*  
 ⟨*proof*⟩

**lemma** *tm3* [*transforms-intros*]:  
**assumes** *ttt* = 16 + *length* zs  
**shows** *transforms tm3 tps0 ttt tps3*  
 ⟨*proof*⟩

**definition** *tps4* :: *tape list* **where**  
*tps4*  $\equiv$  *tps0*  
 [*j2* := ( $\lfloor$ zs = []  $\vee$  last zs = z $\rfloor_B$ , 1)]

**lemma** *tm4*:  
**assumes** *ttt* = 18 + 2 \* *length* zs  
**shows** *transforms tm4 tps0 ttt tps4*  
 ⟨*proof*⟩

**end**

**end**

**lemma** *transforms-tm-empty-or-endswithI* [*transforms-intros*]:  
**fixes** *j1 j2* :: *tapeidx* **and** *z* :: *symbol*  
**fixes** *tps tps'* :: *tape list* **and** *k* :: *nat* **and** *zs* :: *symbol list*  
**assumes** *j1*  $\neq$  *j2* *j1* < *k* *j2* < *k*  
**and** *length* *tps* = *k*  
**and** *proper-symbols* *zs*  
**assumes**  
*tps* ! *j1* = ( $\lfloor$ zs $\rfloor$ , 1)  
*tps* ! *j2* = ( $\lfloor$ 0 $\rfloor_N$ , 1)  
**assumes** *ttt* = 18 + 2 \* *length* *zs*  
**assumes** *tps'* = *tps*  
 [*j2* := ( $\lfloor$ zs = []  $\vee$  last zs = z $\rfloor_B$ , 1)]  
**shows** *transforms (tm-empty-or-endswith j1 j2 z) tps ttt tps'*  
 ⟨*proof*⟩

### 2.11.5 Stripping trailing symbols

Stripping the symbol *z* from the end of a symbol sequence *zs* means:

**definition** *rstrip* :: *symbol*  $\Rightarrow$  *symbol list*  $\Rightarrow$  *symbol list* **where**  
*rstrip* *z* *zs*  $\equiv$  *take* (*LEAST* *i*. *i*  $\leq$  *length* *zs*  $\wedge$  *set* (*drop* *i* *zs*)  $\subseteq$  {*z*}) *zs*

**lemma** *length-rstrip*: *length* (*rstrip* *z* *zs*) = (*LEAST* *i*. *i*  $\leq$  *length* *zs*  $\wedge$  *set* (*drop* *i* *zs*)  $\subseteq$  {*z*})  
 ⟨*proof*⟩

**lemma** *length-rstrip-le*: *length* (*rstrip* *z* *zs*)  $\leq$  *length* *zs*

*<proof>*

**lemma** *rstrip-stripped*:

**assumes**  $i \geq \text{length } (\text{rstrip } z \text{ } zs)$  **and**  $i < \text{length } zs$

**shows**  $zs ! i = z$

*<proof>*

**lemma** *rstrip-replicate*:  $\text{rstrip } z \text{ } (\text{replicate } n \ z) = []$

*<proof>*

**lemma** *rstrip-not-ex*:

**assumes**  $\neg (\exists i < \text{length } zs. zs ! i \neq z)$

**shows**  $\text{rstrip } z \text{ } zs = []$

*<proof>*

**lemma**

**assumes**  $\exists i < \text{length } zs. zs ! i \neq z$

**shows** *rstrip-ex-length*:  $\text{length } (\text{rstrip } z \text{ } zs) > 0$

**and** *rstrip-ex-last*:  $\text{last } (\text{rstrip } z \text{ } zs) \neq z$

*<proof>*

A Turing machine stripping the non-blank, non-start symbol  $z$  from a proper symbol sequence works in the obvious way. First it moves to the end of the symbol sequence, that is, to the first blank. Then it moves left to the first non- $z$  symbol thereby overwriting every symbol with a blank. Finally it performs a “carriage return”.

**definition** *tm-rstrip* :: *symbol*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-rstrip*  $z \ j \equiv$

*tm-right-until*  $j \ \{\square\} \ ;;$

*tm-left*  $j \ ;;$

*tm-lconst-until*  $j \ j \ (\text{UNIV} - \{z\}) \ \square \ ;;$

*tm-cr*  $j$

**lemma** *tm-rstrip-tm*:

**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j$  **and**  $j < k$

**shows** *turing-machine*  $k \ G \ (\text{tm-rstrip } z \ j)$

*<proof>*

**locale** *turing-machine-rstrip* =

**fixes**  $z :: \text{symbol}$  **and**  $j :: \text{tapeidx}$

**begin**

**definition**  $tm1 \equiv \text{tm-right-until } j \ \{\square\}$

**definition**  $tm2 \equiv tm1 \ ;; \text{tm-left } j$

**definition**  $tm3 \equiv tm2 \ ;; \text{tm-lconst-until } j \ j \ (\text{UNIV} - \{z\}) \ \square$

**definition**  $tm4 \equiv tm3 \ ;; \text{tm-cr } j$

**lemma** *tm4-eq-tm-rstrip*:  $tm4 = \text{tm-rstrip } z \ j$

*<proof>*

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $zs :: \text{symbol list}$  **and**  $k :: \text{nat}$

**assumes**  $z : z > 1$

**assumes**  $zs$ : *proper-symbols*  $zs$

**assumes**  $jk$ :  $0 < j \ j < k \ \text{length } tps0 = k$

**assumes**  $tps0$ :  $tps0 ! j = (\lfloor zs \rfloor, 1)$

**begin**

**definition**  $tps1 \equiv tps0$

$[j := (\lfloor zs \rfloor, \text{Suc } (\text{length } zs))]$

**lemma** *tm1* [*transforms-intros*]:

**assumes**  $ttt = \text{Suc } (\text{length } zs)$

**shows** *transforms*  $tm1 \ tps0 \ ttt \ tps1$

*<proof>*

**definition**  $tps2 \equiv tps0$   
[ $j := (\lfloor zs \rfloor, \text{length } zs)$ ]

**lemma**  $tm2$  [transforms-intros]:  
**assumes**  $ttt = \text{length } zs + 2$   
**shows** transforms  $tm2$   $tps0$   $ttt$   $tps2$   
*<proof>*

**definition**  $tps3 \equiv tps0$   
[ $j := (\lfloor rstrip\ z\ zs \rfloor, \text{length } (rstrip\ z\ zs))$ ]

**lemma**  $tm3$  [transforms-intros]:  
**assumes**  $ttt = \text{length } zs + 2 + \text{Suc } (\text{length } zs - \text{length } (rstrip\ z\ zs))$   
**shows** transforms  $tm3$   $tps0$   $ttt$   $tps3$   
*<proof>*

**definition**  $tps4 \equiv tps0$   
[ $j := (\lfloor rstrip\ z\ zs \rfloor, 1)$ ]

**lemma**  $tm4$ :  
**assumes**  $ttt = \text{length } zs + 2 + \text{Suc } (\text{length } zs - \text{length } (rstrip\ z\ zs)) + \text{length } (rstrip\ z\ zs) + 2$   
**shows** transforms  $tm4$   $tps0$   $ttt$   $tps4$   
*<proof>*

**lemma**  $tm4'$ :  
**assumes**  $ttt = 3 * \text{length } zs + 5$   
**shows** transforms  $tm4$   $tps0$   $ttt$   $tps4$   
*<proof>*

**end**

**end**

**lemma**  $transforms-tm-rstripI$  [transforms-intros]:  
**fixes**  $z :: \text{symbol}$  **and**  $j :: \text{tapeidx}$   
**fixes**  $tps\ tps' :: \text{tape list}$  **and**  $zs :: \text{symbol list}$  **and**  $k :: \text{nat}$   
**assumes**  $z > 1$  **and**  $0 < j < k$   
**and** proper-symbols  $zs$   
**and**  $\text{length } tps = k$   
**assumes**  $tps ! j = (\lfloor zs \rfloor, 1)$   
**assumes**  $ttt = 3 * \text{length } zs + 5$   
**assumes**  $tps' = tps[j := (\lfloor rstrip\ z\ zs \rfloor, 1)]$   
**shows** transforms  $(tm-rstrip\ z\ j)$   $tps$   $ttt$   $tps'$   
*<proof>*

## 2.11.6 Writing arbitrary length sequences of the same symbol

The next Turing machine accepts a number  $n$  on tape  $j_1$  and writes the symbol sequence  $z^n$  to tape  $j_2$ . The symbol  $z$  is a parameter of the TM. The TM decrements the number on tape  $j_1$  until it reaches zero.

**definition**  $tm\text{-write-replicate} :: \text{symbol} \Rightarrow \text{tapeidx} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**  
 $tm\text{-write-replicate } z\ j1\ j2 \equiv$   
  WHILE  $\square$  ;  $\lambda rs. rs ! j1 \neq \square$  DO  
     $tm\text{-char } j2\ z$  ;;  
     $tm\text{-decr } j1$   
  DONE ;;  
   $tm\text{-cr } j2$

**lemma**  $tm\text{-write-replicate-tm}$ :  
**assumes**  $0 < j1$  **and**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$  **and**  $j1 \neq j2$  **and**  $G \geq 4$  **and**  $z < G$   
**shows** turing-machine  $k\ G$  ( $tm\text{-write-replicate } z\ j1\ j2$ )  
*<proof>*

**locale** *turing-machine-write-replicate* =  
**fixes**  $j1\ j2 :: \text{tapeidx}$  **and**  $z :: \text{symbol}$   
**begin**

**definition**  $tm1 \equiv tm\text{-char } j2\ z$

**definition**  $tm2 \equiv tm1 \ ;\ ;\ tm\text{-decr } j1$

**definition**  $tmL \equiv \text{WHILE } [] \ ;\ \lambda rs. rs \ !\ j1 \neq \square \ \text{DO } tm2 \ \text{DONE}$

**definition**  $tm3 \equiv tmL \ ;\ ;\ tm\text{-cr } j2$

**lemma**  $tm3\text{-eq-}tm\text{-write-replicate}$ :  $tm3 = tm\text{-write-replicate } z\ j1\ j2$   
 $\langle \text{proof} \rangle$

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $k\ n :: \text{nat}$

**assumes**  $jk$ :  $\text{length } tps0 = k$   $0 < j1$   $0 < j2$   $j1 \neq j2$   $j1 < k$   $j2 < k$   
**and**  $z$ :  $1 < z$

**assumes**  $tps0$ :

$tps0 \ !\ j1 = (\lfloor n \rfloor_N, 1)$

$tps0 \ !\ j2 = (\lfloor [] \rfloor, 1)$

**begin**

**definition**  $tpsL :: \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsL\ t \equiv tps0$

$[j1 := (\lfloor n - t \rfloor_N, 1),$

$j2 := (\lfloor \text{replicate } t\ z \rfloor, \text{Suc } t)]$

**lemma**  $tpsL0$ :  $tpsL\ 0 = tps0$

$\langle \text{proof} \rangle$

**definition**  $tpsL1 :: \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsL1\ t \equiv tps0$

$[j1 := (\lfloor n - t \rfloor_N, 1),$

$j2 := (\lfloor \text{replicate } (\text{Suc } t)\ z \rfloor, \text{Suc } (\text{Suc } t))]$

**lemma**  $tmL1$  [*transforms-intros*]:  $\text{transforms } tm1\ (tpsL\ t)\ 1\ (tpsL1\ t)$

$\langle \text{proof} \rangle$

**lemma**  $tmL2$ :

**assumes**  $ttt = 9 + 2 * \text{nlength } (n - t)$

**shows**  $\text{transforms } tm2\ (tpsL\ t)\ ttt\ (tpsL\ (\text{Suc } t))$

$\langle \text{proof} \rangle$

**lemma**  $tmL2'$  [*transforms-intros*]:

**assumes**  $ttt = 9 + 2 * \text{nlength } n$

**shows**  $\text{transforms } tm2\ (tpsL\ t)\ ttt\ (tpsL\ (\text{Suc } t))$

$\langle \text{proof} \rangle$

**lemma**  $tmL$  [*transforms-intros*]:

**assumes**  $ttt = n * (11 + 2 * \text{nlength } n) + 1$

**shows**  $\text{transforms } tmL\ (tpsL\ 0)\ ttt\ (tpsL\ n)$

$\langle \text{proof} \rangle$

**definition**  $tps3 :: \text{tape list}$  **where**

$tps3 \equiv tps0$

$[j1 := (\lfloor 0 \rfloor_N, 1),$

$j2 := (\lfloor \text{replicate } n\ z \rfloor, 1)]$

**lemma**  $tm3$ :

**assumes**  $ttt = n * (12 + 2 * \text{nlength } n) + 4$

**shows**  $\text{transforms } tm3\ (tpsL\ 0)\ ttt\ tps3$

$\langle \text{proof} \rangle$



**lemma** *length-first*:  $\text{length } (\text{first } ys) \leq \text{length } ys$   
 ⟨proof⟩

**lemma** *length-second-first*:  $\text{length } (\text{second } zs) = \text{length } zs - \text{Suc } (\text{length } (\text{first } zs))$   
 ⟨proof⟩

**lemma** *length-second*:  $\text{length } (\text{second } zs) \leq \text{length } zs$   
 ⟨proof⟩

Our next goal is to show that *first* and *second* really extract the first and second element of a pair.

**lemma** *bindecode-bitenc*:

**fixes**  $x :: \text{string}$   
**shows**  $\text{bindecode } (\text{string-to-symbols } (\text{bitenc } x)) = \text{string-to-symbols } x$   
 ⟨proof⟩

**lemma** *bindecode-string-pair*:

**fixes**  $x u :: \text{string}$   
**shows**  $\text{bindecode } \langle x; u \rangle = \text{string-to-symbols } x @ \text{\#} @ \text{string-to-symbols } u$   
 ⟨proof⟩

**lemma** *first-pair*:

**fixes**  $ys :: \text{symbol list}$  **and**  $x u :: \text{string}$   
**assumes**  $ys = \text{bindecode } \langle x; u \rangle$   
**shows**  $\text{first } ys = \text{string-to-symbols } x$   
 ⟨proof⟩

**lemma** *second-pair*:

**fixes**  $ys :: \text{symbol list}$  **and**  $x u :: \text{string}$   
**assumes**  $ys = \text{bindecode } \langle x; u \rangle$   
**shows**  $\text{second } ys = \text{string-to-symbols } u$   
 ⟨proof⟩

## A Turing machine for extracting the first element

Unlike most other Turing machines, the one in this section is not meant to be reusable, but rather to compute a function, namely the function *first*. For this reason there are no tape index parameters. Instead, the encoded pair is expected on the input tape, and the output is written to the output tape.

**lemma** *bit-symbols-first*:

**assumes**  $ys = \text{bindecode } (\text{string-to-symbols } x)$   
**shows**  $\text{bit-symbols } (\text{first } ys)$   
 ⟨proof⟩

**definition** *tm-first* :: *machine* **where**

$tm\text{-first} \equiv$   
 $tm\text{-right-many } \{0, 1, 2\} ;;$   
 $tm\text{-bindecode } 0\ 2 ;;$   
 $tm\text{-cp-until } 2\ 1 \{\square, |, \#\}$

**lemma** *tm-first-tm*:  $G \geq 6 \implies k \geq 3 \implies \text{turing-machine } k\ G\ tm\text{-first}$   
 ⟨proof⟩

**locale** *turing-machine-fst-pair* =

**fixes**  $k :: \text{nat}$   
**assumes**  $k: k \geq 3$   
**begin**

**definition**  $tm1 \equiv tm\text{-right-many } \{0, 1, 2\}$

**definition**  $tm2 \equiv tm1 ;; tm\text{-bindecode } 0\ 2$

**definition**  $tm3 \equiv tm2 ;; tm\text{-cp-until } 2\ 1 \{\square, |, \#\}$

**lemma** *tm3-eq-tm-first*:  $tm3 = tm\text{-first}$

*<proof>*

**context**

**fixes**  $xs :: \text{symbol list}$

**assumes**  $bs: \text{bit-symbols } xs$

**begin**

**definition**  $tps0 \equiv \text{snd } (\text{start-config } k \ xs)$

**lemma**  $\text{lentps } [simp]: \text{length } tps0 = k$

*<proof>*

**lemma**  $tps0-0: tps0 ! 0 = (\lfloor xs \rfloor, 0)$

*<proof>*

**lemma**  $tps0-gt-0: j > 0 \implies j < k \implies tps0 ! j = (\lfloor [] \rfloor, 0)$

*<proof>*

**definition**  $tps1 \equiv tps0$

$[0 := (\lfloor xs \rfloor, 1),$

$1 := (\lfloor [] \rfloor, 1),$

$2 := (\lfloor [] \rfloor, 1)]$

**lemma**  $tm1$  [*transforms-intros*]: *transforms*  $tm1$   $tps0$   $1$   $tps1$

*<proof>*

**definition**  $tps2 \equiv tps0$

$[0 := (\lfloor xs \rfloor, 1),$

$1 := (\lfloor [] \rfloor, 1),$

$2 := (\lfloor \text{bindecode } xs \rfloor, 1)]$

**lemma**  $tm2$  [*transforms-intros*]:

**assumes**  $ttt = 8 + 3 * \text{length } xs$

**shows** *transforms*  $tm2$   $tps0$   $ttt$   $tps2$

*<proof>*

**definition**  $tps3 \equiv tps0$

$[0 := (\lfloor xs \rfloor, 1),$

$1 := (\lfloor \text{first } (\text{bindecode } xs) \rfloor, \text{Suc } (\text{length } (\text{first } (\text{bindecode } xs))))),$

$2 := (\lfloor \text{bindecode } xs \rfloor, \text{Suc } (\text{length } (\text{first } (\text{bindecode } xs))))]$

**lemma**  $tm3$ :

**assumes**  $ttt = 8 + 3 * \text{length } xs + \text{Suc } (\text{length } (\text{first } (\text{bindecode } xs)))$

**shows** *transforms*  $tm3$   $tps0$   $ttt$   $tps3$

*<proof>*

**lemma**  $tm3'$ :

**assumes**  $ttt = 9 + 4 * \text{length } xs$

**shows** *transforms*  $tm3$   $tps0$   $ttt$   $tps3$

*<proof>*

**end**

**lemma**  $tm3$ -computes:

*computes-in-time*  $k$   $tm3$   $(\lambda x. \text{symbols-to-string } (\text{first } (\text{bindecode } (\text{string-to-symbols } x))))$   $(\lambda n. 9 + 4 * n)$

*<proof>*

**end**

**lemma**  $tm$ -first-computes:

**assumes**  $k \geq 3$

**shows** *computes-in-time*

$k$

```

tm-first
(λx. symbols-to-string (first (bindecode (string-to-symbols x))))
(λn. 9 + 4 * n)
⟨proof⟩

```

## A Turing machine for splitting pairs

The next Turing machine expects a proper symbol sequence  $zs$  on tape  $j_1$  and outputs *first*  $zs$  and *second*  $zs$  on tapes  $j_2$  and  $j_3$ , respectively.

**definition**  $tm\text{-}unpair :: tapeidx \Rightarrow tapeidx \Rightarrow tapeidx \Rightarrow machine$  **where**

```

tm-unpair j1 j2 j3 ≡
tm-cp-until j1 j2 {□, |, #} ;;
tm-right j1 ;;
tm-cp-until j1 j3 {□} ;;
tm-cr j1 ;;
tm-cr j2 ;;
tm-cr j3

```

**lemma**  $tm\text{-}unpair\text{-}tm$ :

```

assumes  $k \geq 2$  and  $G \geq 4$  and  $0 < j_2$  and  $0 < j_3$  and  $j_1 < k$   $j_2 < k$   $j_3 < k$ 
shows  $turing\text{-}machine\ k\ G\ (tm\text{-}unpair\ j_1\ j_2\ j_3)$ 
⟨proof⟩

```

**locale**  $turing\text{-}machine\text{-}unpair =$

```

fixes  $j_1\ j_2\ j_3 :: tapeidx$ 

```

**begin**

**definition**  $tm1 \equiv tm\text{-}cp\text{-}until\ j_1\ j_2\ \{\square, |, \#\}$

**definition**  $tm2 \equiv tm1\ \;;\ tm\text{-}right\ j_1$

**definition**  $tm3 \equiv tm2\ \;;\ tm\text{-}cp\text{-}until\ j_1\ j_3\ \{\square\}$

**definition**  $tm4 \equiv tm3\ \;;\ tm\text{-}cr\ j_1$

**definition**  $tm5 \equiv tm4\ \;;\ tm\text{-}cr\ j_2$

**definition**  $tm6 \equiv tm5\ \;;\ tm\text{-}cr\ j_3$

**lemma**  $tm6\text{-}eq\text{-}tm\text{-}unpair$ :  $tm6 = tm\text{-}unpair\ j_1\ j_2\ j_3$

⟨proof⟩

**context**

```

fixes  $tps0 :: tape\ list$  and  $k :: nat$  and  $zs :: symbol\ list$ 

```

```

assumes  $jk$ :  $0 < j_2$   $0 < j_3$   $j_1 \neq j_2$   $j_1 \neq j_3$   $j_2 \neq j_3$   $j_1 < k$   $j_2 < k$   $j_3 < k$   $length\ tps0 = k$ 

```

```

and  $zs$ :  $proper\text{-}symbols\ zs$ 

```

```

and  $tps0$ :

```

```

 $tps0\ !\ j_1 = (\lfloor zs \rfloor, 1)$ 

```

```

 $tps0\ !\ j_2 = (\lfloor \square \rfloor, 1)$ 

```

```

 $tps0\ !\ j_3 = (\lfloor \square \rfloor, 1)$ 

```

**begin**

**definition**  $tps1 \equiv tps0$

```

 $[j_1 := (\lfloor zs \rfloor, Suc\ (length\ (first\ zs))),$ 

```

```

 $j_2 := (\lfloor first\ zs \rfloor, Suc\ (length\ (first\ zs)))]$ 

```

**lemma**  $tm1$  [transforms-intros]:

```

assumes  $ttt = Suc\ (length\ (first\ zs))$ 

```

```

shows  $transforms\ tm1\ tps0\ ttt\ tps1$ 

```

⟨proof⟩

**definition**  $tps2 \equiv tps0$

```

 $[j_1 := (\lfloor zs \rfloor, length\ (first\ zs) + 2),$ 

```

```

 $j_2 := (\lfloor first\ zs \rfloor, Suc\ (length\ (first\ zs)))]$ 

```

**lemma**  $tm2$  [transforms-intros]:

```

assumes  $ttt = length\ (first\ zs) + 2$ 

```

```

shows  $transforms\ tm2\ tps0\ ttt\ tps2$ 

```

*<proof>*

**definition**  $tps3 \equiv tps0$

$[j1 := (\lfloor zs \rfloor, \text{length}(\text{first } zs) + 2 + (\text{length } zs - \text{Suc}(\text{length}(\text{first } zs))))],$   
 $j2 := (\lfloor \text{first } zs \rfloor, \text{Suc}(\text{length}(\text{first } zs))),$   
 $j3 := (\lfloor \text{second } zs \rfloor, \text{Suc}(\text{length}(\text{second } zs)))]$

**lemma**  $tm3$  [transforms-intros]:

**assumes**  $ttt = \text{length}(\text{first } zs) + 2 + \text{Suc}(\text{length } zs - \text{Suc}(\text{length}(\text{first } zs)))$

**shows**  $\text{transforms } tm3 \ tps0 \ ttt \ tps3$

*<proof>*

**definition**  $tps4 \equiv tps0$

$[j1 := (\lfloor zs \rfloor, 1),$   
 $j2 := (\lfloor \text{first } zs \rfloor, \text{Suc}(\text{length}(\text{first } zs))),$   
 $j3 := (\lfloor \text{second } zs \rfloor, \text{Suc}(\text{length}(\text{second } zs)))]$

**lemma**  $tm4$ :

**assumes**  $ttt = 2 * \text{length}(\text{first } zs) + 7 + 2 * (\text{length } zs - \text{Suc}(\text{length}(\text{first } zs)))$

**shows**  $\text{transforms } tm4 \ tps0 \ ttt \ tps4$

*<proof>*

**lemma**  $tm4'$  [transforms-intros]:

**assumes**  $ttt = 4 * \text{length } zs + 7$

**shows**  $\text{transforms } tm4' \ tps0 \ ttt \ tps4$

*<proof>*

**definition**  $tps5 \equiv tps0$

$[j1 := (\lfloor zs \rfloor, 1),$   
 $j2 := (\lfloor \text{first } zs \rfloor, 1),$   
 $j3 := (\lfloor \text{second } zs \rfloor, \text{Suc}(\text{length}(\text{second } zs)))]$

**lemma**  $tm5$  [transforms-intros]:

**assumes**  $ttt = 4 * \text{length } zs + 9 + \text{Suc}(\text{length}(\text{first } zs))$

**shows**  $\text{transforms } tm5 \ tps0 \ ttt \ tps5$

*<proof>*

**definition**  $tps6 \equiv tps0$

$[j1 := (\lfloor zs \rfloor, 1),$   
 $j2 := (\lfloor \text{first } zs \rfloor, 1),$   
 $j3 := (\lfloor \text{second } zs \rfloor, 1)]$

**lemma**  $tm6$ :

**assumes**  $ttt = 4 * \text{length } zs + 11 + \text{Suc}(\text{length}(\text{first } zs)) + \text{Suc}(\text{length}(\text{second } zs))$

**shows**  $\text{transforms } tm6 \ tps0 \ ttt \ tps6$

*<proof>*

**definition**  $tps6' \equiv tps0$

$[j2 := (\lfloor \text{first } zs \rfloor, 1),$   
 $j3 := (\lfloor \text{second } zs \rfloor, 1)]$

**lemma**  $tps6'$ :  $tps6' = tps6$

*<proof>*

**lemma**  $tm6'$ :

**assumes**  $ttt = 6 * \text{length } zs + 13$

**shows**  $\text{transforms } tm6' \ tps0 \ ttt \ tps6'$

*<proof>*

**end**

**end**

```

lemma transforms-tm-unpairI [transforms-intros]:
  fixes j1 j2 j3 :: tapeidx
  fixes tps tps' :: tape list and k :: nat and zs :: symbol list
  assumes  $0 < j2$   $0 < j3$   $j1 \neq j2$   $j1 \neq j3$   $j2 \neq j3$   $j1 < k$   $j2 < k$   $j3 < k$ 
    and length tps = k
    and proper-symbols zs
  assumes
    tps ! j1 = ([zs], 1)
    tps ! j2 = ([[]], 1)
    tps ! j3 = ([[]], 1)
  assumes ttt = 6 * length zs + 13
  assumes tps' = tps
    [j2 := ([first zs], 1),
     j3 := ([second zs], 1)]
  shows transforms (tm-unpair j1 j2 j3) tps ttt tps'
  <proof>

end

```

## 2.12 Well-formedness of lists

```

theory Wellformed
  imports Symbol-Ops Lists-Lists
begin

```

In the representations introduced in Section 2.8 and Section 2.9, not every symbol sequence over  $\mathbf{01|}$  represents a list of numbers, and not every symbol sequence over  $\mathbf{01|}\#$  represents a list of lists of numbers. In this section we prove criteria for symbol sequences to represent such lists and devise Turing machines to check these criteria efficiently.

### 2.12.1 A criterion for well-formed lists

From the definition of *numlist* it is easy to see that a symbol sequence representing a list of numbers is either empty or not, and that in the latter case it ends with a  $|$  symbol. Moreover it can only contain the symbols  $\mathbf{01|}$  and cannot contain the symbol sequence  $\mathbf{0|}$  because canonical number representations cannot end in  $\mathbf{0}$ . That these properties are not only necessary but also sufficient for the symbol sequence to represent a list of numbers is shown in this section.

A symbol sequence is well-formed if it represents a list of numbers.

```

definition numlist-wf :: symbol list  $\Rightarrow$  bool where
  numlist-wf zs  $\equiv$   $\exists ns. numlist ns = zs$ 
```

```

lemma numlist-wf-append:
  assumes numlist-wf xs and numlist-wf ys
  shows numlist-wf (xs @ ys)
  <proof>

```

```

lemma numlist-wf-canonical:
  assumes canonical xs
  shows numlist-wf (xs @ [[]])
  <proof>

```

Well-formed symbol sequences can be unambiguously decoded to lists of numbers.

```

definition zs-numlist :: symbol list  $\Rightarrow$  nat list where
  zs-numlist zs  $\equiv$  THE ns. numlist ns = zs

```

```

lemma zs-numlist-ex1:
  assumes numlist-wf zs
  shows  $\exists! ns.$  numlist ns = zs
  <proof>

```

```

lemma numlist-zs-numlist:

```

**assumes** *numlist-wf zs*  
**shows** *numlist (zs-numlist zs) = zs*  
 ⟨*proof*⟩

Count the number of occurrences of an element in a list:

**fun** *count* :: *nat list* ⇒ *nat* ⇒ *nat* **where**  
*count* [] *z* = 0 |  
*count* (*x # xs*) *z* = (if *x = z* then 1 else 0) + *count xs z*

**lemma** *count-append*: *count (xs @ ys) z = count xs z + count ys z*  
 ⟨*proof*⟩

**lemma** *count-0*: *count xs z = 0* ⇔ (∀ *x* ∈ *set xs*. *x* ≠ *z*)  
 ⟨*proof*⟩

**lemma** *count-gr-0-take*:  
**assumes** *count xs z > 0*  
**shows** ∃ *j*.  
*j* < *length xs* ∧  
*xs* ! *j* = *z* ∧  
 (∀ *i* < *j*. *xs* ! *i* ≠ *z*) ∧  
*count (take (Suc *j*) xs) z = 1* ∧  
*count (drop (Suc *j*) xs) z = count xs z - 1*  
 ⟨*proof*⟩

**definition** *has2* :: *symbol list* ⇒ *symbol* ⇒ *symbol* ⇒ *bool* **where**  
*has2 xs y z* ≡ ∃ *i* < *length xs* - 1. *xs* ! *i* = *y* ∧ *xs* ! (Suc *i*) = *z*

**lemma** *not-has2-take*:  
**assumes** ¬ *has2 xs y z*  
**shows** ¬ *has2 (take m xs) y z*  
 ⟨*proof*⟩

**lemma** *not-has2-drop*:  
**assumes** ¬ *has2 xs y z*  
**shows** ¬ *has2 (drop m xs) y z*  
 ⟨*proof*⟩

**lemma** *numlist-wf-has2*:  
**assumes** *proper-symbols xs symbols-lt 5 xs* ¬ *has2 xs 0* | *xs* ≠ [] → *last xs* = |  
**shows** *numlist-wf xs*  
 ⟨*proof*⟩

**lemma** *last-numlist-4*: *numlist ns* ≠ [] ⇒ *last (numlist ns)* = |  
 ⟨*proof*⟩

**lemma** *numlist-not-has2*:  
**assumes** *i* < *length (numlist ns)* - 1 **and** *numlist ns* ! *i* = 0  
**shows** *numlist ns* ! (Suc *i*) ≠ |  
 ⟨*proof*⟩

**lemma** *numlist-wf-has2'*:  
**assumes** *numlist-wf xs*  
**shows** *proper-symbols-lt 5 xs* ∧ ¬ *has2 xs 0* | ∧ (*xs* ≠ [] → *last xs* = |)  
 ⟨*proof*⟩

**lemma** *numlist-wf-iff*:  
*numlist-wf xs* ⇔ *proper-symbols-lt 5 xs* ∧ ¬ *has2 xs 0* | ∧ (*xs* ≠ [] → *last xs* = |)  
 ⟨*proof*⟩

## 2.12.2 A criterion for well-formed lists of lists

The criterion for lists of lists of numbers is similar to the one for lists of numbers. A non-empty symbol sequence must end in  $\#$ . All symbols must be from  $\mathbf{01}\#$  and the sequences  $\mathbf{0}$ ,  $\mathbf{0}\#$ , and  $\mathbf{1}\#$  are forbidden. A symbol sequence is well-formed if it represents a list of lists of numbers.

**definition** *numlistlist-wf* :: *symbol list*  $\Rightarrow$  *bool* **where**  
*numlistlist-wf* *zs*  $\equiv \exists$  *nss*. *numlistlist* *nss* = *zs*

**lemma** *numlistlist-wf-append*:  
**assumes** *numlistlist-wf* *xs* **and** *numlistlist-wf* *ys*  
**shows** *numlistlist-wf* (*xs* @ *ys*)  
*<proof>*

**lemma** *numlistlist-wf-numlist-wf*:  
**assumes** *numlist-wf* *xs*  
**shows** *numlistlist-wf* (*xs* @ [#])  
*<proof>*

**lemma** *numlistlist-wf-has2*:  
**assumes** *proper-symbols* *xs* *symbols-lt* 6 *xs* *xs*  $\neq [] \longrightarrow$  *last* *xs* =  $\#$   
**and**  $\neg$  *has2* *xs*  $\mathbf{0}$  |  
**and**  $\neg$  *has2* *xs*  $\mathbf{0}\#$   
**and**  $\neg$  *has2* *xs*  $\mathbf{1}\#$   
**shows** *numlistlist-wf* *xs*  
*<proof>*

**lemma** *numlistlist-not-has2*:  
**assumes**  $i < \text{length} (\text{numlistlist } nss) - 1$  **and** *numlistlist* *nss* !  $i = \mathbf{0}$   
**shows** *numlistlist* *nss* ! (*Suc*  $i$ )  $\neq \#$  |  
*<proof>*

**lemma** *numlistlist-not-has2'*:  
**assumes**  $i < \text{length} (\text{numlistlist } nss) - 1$  **and** *numlistlist* *nss* !  $i = \mathbf{0} \vee$  *numlistlist* *nss* !  $i = \mathbf{1}$   
**shows** *numlistlist* *nss* ! (*Suc*  $i$ )  $\neq \#$   
*<proof>*

**lemma** *last-numlistlist-5*: *numlistlist* *nss*  $\neq [] \Longrightarrow$  *last* (*numlistlist* *nss*) =  $\#$   
*<proof>*

**lemma** *numlistlist-wf-has2'*:  
**assumes** *numlistlist-wf* *xs*  
**shows** *proper-symbols-lt* 6 *xs*  $\wedge$  (*xs*  $\neq [] \longrightarrow$  *last* *xs* =  $\#$ )  $\wedge$   $\neg$  *has2* *xs*  $\mathbf{0}$  |  $\wedge$   $\neg$  *has2* *xs*  $\mathbf{0}\#$   $\wedge$   $\neg$  *has2* *xs*  $\mathbf{1}\#$   
*<proof>*

**lemma** *numlistlist-wf-iff*:  
*numlistlist-wf* *xs*  $\longleftrightarrow$   
*proper-symbols-lt* 6 *xs*  $\wedge$  (*xs*  $\neq [] \longrightarrow$  *last* *xs* =  $\#$ )  $\wedge$   $\neg$  *has2* *xs*  $\mathbf{0}$  |  $\wedge$   $\neg$  *has2* *xs*  $\mathbf{0}\#$   $\wedge$   $\neg$  *has2* *xs*  $\mathbf{1}\#$   
*<proof>*

## 2.12.3 A Turing machine to check for subsequences of length two

In order to implement the well-formedness criteria we need to be able to check a symbol sequence for subsequences of length two. The next Turing machine has symbol parameters *y* and *z* and checks whether the sequence [*y*, *z*] exists on tape  $j_1$ . It writes to tape  $j_2$  the number 0 or 1 if the sequence is present or not, respectively.

**definition** *tm-not-has2* :: *symbol*  $\Rightarrow$  *symbol*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**  
*tm-not-has2* *y* *z*  $j_1$   $j_2 \equiv$   
*tm-set*  $j_2$  [ $\mathbf{0}$ ,  $\mathbf{0}$ ] ;;  
**WHILE** [] ;  $\lambda$ *rs*. *rs* !  $j_1 \neq \square$  **DO**  
**IF**  $\lambda$ *rs*. *rs* !  $j_2 = \mathbf{1} \wedge$  *rs* !  $j_1 = z$  **THEN**  
*tm-right*  $j_2$  ;;  
*tm-write*  $j_2$   $\mathbf{1}$  ;;

```

    tm-left j2
  ELSE
    []
  ENDIF ;;
  tm-trans2 j1 j2 ( $\lambda h. \text{if } h = y \text{ then } \mathbf{1} \text{ else } \mathbf{0}$ ) ;;
  tm-right j1
  DONE ;;
  tm-right j2 ;;
  IF  $\lambda rs. rs ! j2 = \mathbf{1}$  THEN
    tm-set j2 (canrepr 1)
  ELSE
    tm-set j2 (canrepr 0)
  ENDIF ;;
  tm-cr j1 ;;
  tm-not j2

```

**lemma** *tm-not-has2-tm*:

**assumes**  $k \geq 2$  **and**  $G \geq 4$  **and**  $0 < j2$  **and**  $j1 < k$  **and**  $j2 < k$   
**shows** *turing-machine*  $k$   $G$  (*tm-not-has2*  $y$   $z$   $j1$   $j2$ )  
*<proof>*

**locale** *turing-machine-has2* =

**fixes**  $y$   $z$  :: *symbol* **and**  $j1$   $j2$  :: *tapeidx*  
**begin**

**context**

**fixes**  $tps0$  :: *tape list* **and**  $xs$  :: *symbol list* **and**  $k$  :: *nat*  
**assumes**  $xs$ : *proper-symbols*  $xs$   
**assumes**  $yz$ :  $y > 1$   $z > 1$   
**assumes**  $jk$ :  $j1 < k$   $j2 < k$   $j1 \neq j2$   $0 < j2$   $\text{length } tps0 = k$   
**assumes**  $tps0$ :  
 $tps0 ! j1 = ([xs], 1)$   
 $tps0 ! j2 = ([[]], 1)$

**begin**

**definition**  $tm1 \equiv tm\text{-set } j2$   $[0, 0]$

**definition**  $tmT1 \equiv tm\text{-right } j2$

**definition**  $tmT2 \equiv tmT1$  ;; *tm-write*  $j2$   $\mathbf{1}$

**definition**  $tmT3 \equiv tmT2$  ;; *tm-left*  $j2$

**definition**  $tmL1 \equiv IF \lambda rs. rs ! j2 = \mathbf{1} \wedge rs ! j1 = z THEN tmT3 ELSE [] ENDIF$

**definition**  $tmL2 \equiv tmL1$  ;; *tm-trans2*  $j1$   $j2$  ( $\lambda h. \text{if } h = y \text{ then } \mathbf{1} \text{ else } \mathbf{0}$ )

**definition**  $tmL3 \equiv tmL2$  ;; *tm-right*  $j1$

**definition**  $tmL \equiv WHILE [] ; \lambda rs. rs ! j1 \neq \square DO tmL3 DONE$

**definition**  $tm2 \equiv tm1$  ;;  $tmL$

**definition**  $tm3 \equiv tm2$  ;; *tm-right*  $j2$

**definition**  $tm34 \equiv IF \lambda rs. rs ! j2 = \mathbf{1} THEN tm\text{-set } j2$  (canrepr 1) ELSE  $tm\text{-set } j2$  (canrepr 0) ENDIF

**definition**  $tm4 \equiv tm3$  ;;  $tm34$

**definition**  $tm5 \equiv tm4$  ;; *tm-cr*  $j1$

**definition**  $tm6 \equiv tm5$  ;; *tm-not*  $j2$

**lemma** *tm6-eq-tm-not-has2*:  $tm6 = tm\text{-not-has2 } y$   $z$   $j1$   $j2$

*<proof>*

**definition**  $tps1$  :: *tape list* **where**

$tps1 \equiv tps0$   
 $[j1] := ([xs], 1),$   
 $[j2] := ([[0, 0]], 1)$

**lemma** *tm1*: *transforms*  $tm1$   $tps0$   $14$   $tps1$

*<proof>*

**abbreviation** *has-at* :: *nat* ⇒ *bool* **where**

*has-at i* ≡ *xs ! i = y* ∧ *xs ! Suc i = z*

**definition** *tpsL* :: *nat* ⇒ *tape list* **where**

*tpsL t* ≡ *tps0*

*j1* := (*[xs]*, *Suc t*),

*j2* := ([*if [xs] t = y then 1 else 0, if ∃ i < t - 1. has-at i then 1 else 0*]], 1))

**lemma** *tpsL-eq-tps1*: *tpsL 0 = tps1*

⟨*proof*⟩

**lemma** *tm1'* [*transforms-intros*]: *transforms tm1 tps0 14 (tpsL 0)*

⟨*proof*⟩

**definition** *tpsT1* :: *nat* ⇒ *tape list* **where**

*tpsT1 t* ≡ *tps0*

*j1* := (*[xs]*, *Suc t*),

*j2* := ([*if [xs] t = y then 1 else 0, if ∃ i < t - 1. has-at i then 1 else 0*]], 2))

**definition** *tpsT2* :: *nat* ⇒ *tape list* **where**

*tpsT2 t* ≡ *tps0*

*j1* := (*[xs]*, *Suc t*),

*j2* := ([*if [xs] t = y then 1 else 0, if ∃ i < t. has-at i then 1 else 0*]], 2))

**definition** *tpsT3* :: *nat* ⇒ *tape list* **where**

*tpsT3 t* ≡ *tps0*

*j1* := (*[xs]*, *Suc t*),

*j2* := ([*if [xs] t = y then 1 else 0, if ∃ i < t. has-at i then 1 else 0*]], 1))

**lemma** *contents-1-update*: ([*[a, b]*], 1) |=| *v = ([v, b]*], 1) **for** *a b v* :: *symbol*

⟨*proof*⟩

**lemma** *contents-2-update*: ([*[a, b]*], 2) |=| *v = ([a, v]*], 2) **for** *a b v* :: *symbol*

⟨*proof*⟩

**context**

**fixes** *t* :: *nat*

**assumes** *then-branch*: [*xs*] *t = y xs ! t = z*

**begin**

**lemma** *tmT1* [*transforms-intros*]: *transforms tmT1 (tpsL t) 1 (tpsT1 t)*

⟨*proof*⟩

**lemma** *tmT2* [*transforms-intros*]: *transforms tmT2 (tpsL t) 2 (tpsT2 t)*

⟨*proof*⟩

**lemma** *tmT3* [*transforms-intros*]: *transforms tmT3 (tpsL t) 3 (tpsT3 t)*

⟨*proof*⟩

**end**

**lemma** *tmL1* [*transforms-intros*]:

**assumes** *ttt = 5* **and** *t < length xs*

**shows** *transforms tmL1 (tpsL t) ttt (tpsT3 t)*

⟨*proof*⟩

**definition** *tpsL2* :: *nat* ⇒ *tape list* **where**

*tpsL2 t* ≡ *tps0*

*j1* := (*[xs]*, *Suc t*),

*j2* := ([*if [xs] (Suc t) = y then 1 else 0, if ∃ i < t. has-at i then 1 else 0*]], 1))

**lemma** *tmL2* [*transforms-intros*]:

**assumes**  $ttt = 6$  **and**  $t < \text{length } xs$   
**shows** *transforms tmL2* (tpsL t) ttt (tpsL2 t)  
⟨proof⟩

**lemma** *tmL3* [*transforms-intros*]:  
**assumes**  $ttt = 7$  **and**  $t < \text{length } xs$   
**shows** *transforms tmL3* (tpsL t) ttt (tpsL (Suc t))  
⟨proof⟩

**lemma** *tmL* [*transforms-intros*]:  
**assumes**  $ttt = 9 * \text{length } xs + 1$   
**shows** *transforms tmL* (tpsL 0) ttt (tpsL (length xs))  
⟨proof⟩

**lemma** *tm2* [*transforms-intros*]:  
**assumes**  $ttt = 9 * \text{length } xs + 15$   
**shows** *transforms tm2* tps0 ttt (tpsL (length xs))  
⟨proof⟩

**definition** *tps3* :: *tape list where*  
 $tps3 \equiv tps0$   
 $[j1 := (\lfloor xs \rfloor, \text{Suc } (\text{length } xs)),$   
 $j2 := ([\text{if } \lfloor xs \rfloor (\text{length } xs) = y \text{ then } \mathbf{1} \text{ else } \mathbf{0}, \text{if } \exists i < \text{length } xs - 1. \text{has-at } i \text{ then } \mathbf{1} \text{ else } \mathbf{0}], 2)]$

**lemma** *tm3* [*transforms-intros*]:  
**assumes**  $ttt = 9 * \text{length } xs + 16$   
**shows** *transforms tm3* tps0 ttt tps3  
⟨proof⟩

**definition** *tps4* :: *tape list where*  
 $tps4 \equiv tps0$   
 $[j1 := (\lfloor xs \rfloor, \text{Suc } (\text{length } xs)),$   
 $j2 := ([\exists i < \text{length } xs - \text{Suc } 0. \text{has-at } i]_B, 1)]$

**lemma** *tm34* [*transforms-intros*]:  
**assumes**  $ttt = 19$   
**shows** *transforms tm34* tps3 ttt tps4  
⟨proof⟩

**lemma** *tm4*:  
**assumes**  $ttt = 9 * \text{length } xs + 35$   
**shows** *transforms tm4* tps0 ttt tps4  
⟨proof⟩

**definition** *tps4'* :: *tape list where*  
 $tps4' \equiv tps0$   
 $[j1 := (\lfloor xs \rfloor, \text{Suc } (\text{length } xs)),$   
 $j2 := ([\text{has2 } xs \ y \ z]_B, 1)]$

**lemma** *tps4'*:  $tps4 = tps4'$   
⟨proof⟩

**lemma** *tm4'* [*transforms-intros*]:  
**assumes**  $ttt = 9 * \text{length } xs + 35$   
**shows** *transforms tm4'* tps0 ttt tps4'  
⟨proof⟩

**definition** *tps5* :: *tape list where*  
 $tps5 \equiv tps0$   
 $[j1 := (\lfloor xs \rfloor, 1),$   
 $j2 := ([\text{has2 } xs \ y \ z]_B, 1)]$

**lemma** *tm5*:

```

assumes  $ttt = 10 * \text{length } xs + 38$ 
shows transforms tm5 tps0 ttt tps5
<proof>

```

**definition** *tps5'* :: *tape list* **where**

```

tps5'  $\equiv$  tps0
[j2 := ( $\lfloor \text{has2 } xs \ y \ z \rfloor_B, 1$ )]

```

**lemma** *tm5'* [*transforms-intros*]:

```

assumes  $ttt = 10 * \text{length } xs + 38$ 
shows transforms tm5 tps0 ttt tps5'
<proof>

```

**definition** *tps6* :: *tape list* **where**

```

tps6  $\equiv$  tps0
[j2 := ( $\lfloor \neg \text{has2 } xs \ y \ z \rfloor_B, 1$ )]

```

**lemma** *tm6*:

```

assumes  $ttt = 10 * \text{length } xs + 41$ 
shows transforms tm6 tps0 ttt tps6
<proof>

```

**end**

**end**

**lemma** *transforms-tm-not-has2I* [*transforms-intros*]:

```

fixes  $y \ z :: \text{symbol}$  and  $j1 \ j2 :: \text{tapeidx}$ 
fixes tps tps' :: tape list and  $xs :: \text{symbol list}$  and  $ttt \ k :: \text{nat}$ 
assumes  $j1 < k \ j2 < k \ j1 \neq j2 \ 0 < j2 \ \text{length } tps = k \ y > 1 \ z > 1$ 
and proper-symbols xs
assumes
   $tps ! j1 = (\lfloor xs \rfloor, 1)$ 
   $tps ! j2 = (\lfloor [] \rfloor, 1)$ 
assumes  $ttt = 10 * \text{length } xs + 41$ 
assumes  $tps' = tps$ 
  [j2 := ( $\lfloor \neg \text{has2 } xs \ y \ z \rfloor_B, 1$ )]
shows transforms (tm-not-has2 y z j1 j2) tps ttt tps'
<proof>

```

## 2.12.4 Checking well-formedness for lists

The next Turing machine checks all conditions from the criterion in lemma *numlist-wf-iff* one after another for the symbols on tape  $j_1$  and writes to tape  $j_2$  either the number 1 or 0 depending on whether all conditions were met. It assumes tape  $j_2$  is initialized with 0.

**definition** *tm-numlist-wf* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-numlist-wf  $j1 \ j2 \equiv$ 
  tm-proper-symbols-lt  $j1 \ j2 \ 5$  ;;
  tm-not-has2  $\mathbf{0} \mid j1 \ (j2 + 1)$  ;;
  tm-and  $j2 \ (j2 + 1)$  ;;
  tm-setn  $(j2 + 1) \ 0$  ;;
  tm-empty-or-endswith  $j1 \ (j2 + 1) \mid$  ;;
  tm-and  $j2 \ (j2 + 1)$  ;;
  tm-setn  $(j2 + 1) \ 0$ 

```

**lemma** *tm-numlist-wf-tm*:

```

assumes  $k \geq 2$  and  $G \geq 5$  and  $0 < j2 \ 0 < j1$  and  $j1 < k \ j2 + 1 < k$ 
shows turing-machine  $k \ G \ (\text{tm-numlist-wf } j1 \ j2)$ 
<proof>

```

**locale** *turing-machine-numlist-wf* =

```

fixes  $j1 \ j2 :: \text{tapeidx}$ 
begin

```

```

definition tm1 ≡ tm-proper-symbols-lt j1 j2 5
definition tm2 ≡ tm1 ;; tm-not-has2 0 | j1 (j2 + 1)
definition tm3 ≡ tm2 ;; tm-and j2 (j2 + 1)
definition tm4 ≡ tm3 ;; tm-setn (j2 + 1) 0
definition tm5 ≡ tm4 ;; tm-empty-or-endswith j1 (j2 + 1) |
definition tm6 ≡ tm5 ;; tm-and j2 (j2 + 1)
definition tm7 ≡ tm6 ;; tm-setn (j2 + 1) 0

lemma tm7-eq-tm-numlist-wf: tm7 = tm-numlist-wf j1 j2
  ⟨proof⟩

context
  fixes tps0 :: tape list and zs :: symbol list and k :: nat
  assumes zs: proper-symbols zs
  assumes jk: 0 < j1 j1 < k j2 + 1 < k j1 ≠ j2 0 < j2 j1 ≠ j2 + 1 length tps0 = k
  assumes tps0:
    tps0 ! j1 = ([zs], 1)
    tps0 ! j2 = ([[]], 1)
    tps0 ! (j2 + 1) = ([[]], 1)
begin

definition tps1 ≡ tps0
  [j2 := ([proper-symbols-lt 5 zs]B, 1)]

lemma tm1 [transforms-intros]:
  assumes ttt = 5 + 7 * length zs
  shows transforms tm1 tps0 ttt tps1
  ⟨proof⟩

definition tps2 ≡ tps0
  [j2 := ([proper-symbols-lt 5 zs]B, 1),
  j2 + 1 := ([if has2 zs 0 | then 0 else 1]N, 1)]

lemma tm2 [transforms-intros]:
  assumes ttt = 46 + 17 * length zs
  shows transforms tm2 tps0 ttt tps2
  ⟨proof⟩

definition tps3 ≡ tps0
  [j2 := ([proper-symbols-lt 5 zs ∧ ¬ has2 zs 0]B, 1),
  j2 + 1 := ([if has2 zs 0 | then 0 else 1]N, 1)]

lemma tm3 [transforms-intros]:
  assumes ttt = 46 + 17 * length zs + 3
  shows transforms tm3 tps0 ttt tps3
  ⟨proof⟩

definition tps4 ≡ tps0
  [j2 := ([proper-symbols-lt 5 zs ∧ ¬ has2 zs 0]B, 1),
  j2 + 1 := ([0]N, 1)]

lemma tm4:
  assumes ttt = 46 + 17 * length zs + 13 + 2 * nlength (if has2 zs 0 | then 0 else 1)
  shows transforms tm4 tps0 ttt tps4
  ⟨proof⟩

lemma tm4' [transforms-intros]:
  assumes ttt = 46 + 17 * length zs + 15
  shows transforms tm4 tps0 ttt tps4
  ⟨proof⟩

definition tps5 ≡ tps0

```

$[j2 := (\lfloor \text{proper-symbols-}lt\ 5\ zs \wedge \neg \text{has2}\ zs\ \mathbf{0} \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor zs = [] \vee \text{last}\ zs = | \rfloor_B, 1)]$

**lemma** *tm5* [*transforms-intros*]:  
**assumes**  $ttt = 79 + 19 * \text{length}\ zs$   
**shows** *transforms tm5 tps0 ttt tps5*  
*<proof>*

**definition** *tps6*  $\equiv tps0$   
 $[j2 := (\lfloor \text{proper-symbols-}lt\ 5\ zs \wedge \neg \text{has2}\ zs\ \mathbf{0} \mid \wedge (zs = [] \vee \text{last}\ zs = |) \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor zs = [] \vee \text{last}\ zs = | \rfloor_B, 1)]$

**lemma** *tm6* [*transforms-intros*]:  
**assumes**  $ttt = 82 + 19 * \text{length}\ zs$   
**shows** *transforms tm6 tps0 ttt tps6*  
*<proof>*

**definition** *tps7*  $\equiv tps0$   
 $[j2 := (\lfloor \text{proper-symbols-}lt\ 5\ zs \wedge \neg \text{has2}\ zs\ \mathbf{0} \mid \wedge (zs = [] \vee \text{last}\ zs = |) \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor 0 \rfloor_N, 1)]$

**lemma** *tm7*:  
**assumes**  $ttt = 92 + 19 * \text{length}\ zs + 2 * \text{nlength}\ (if\ zs = [] \vee \text{last}\ zs = | \text{ then } 1 \text{ else } 0)$   
**shows** *transforms tm7 tps0 ttt tps7*  
*<proof>*

**definition** *tps7'*  $\equiv tps0$   
 $[j2 := (\lfloor \text{numlist-wf}\ zs \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor 0 \rfloor_N, 1)]$

**lemma** *tm7'*:  
**assumes**  $ttt = 94 + 19 * \text{length}\ zs$   
**shows** *transforms tm7 tps0 ttt tps7'*  
*<proof>*

**definition** *tps7''*  $\equiv tps0$   
 $[j2 := (\lfloor \text{numlist-wf}\ zs \rfloor_B, 1)]$

**lemma** *tm7''* [*transforms-intros*]:  
**assumes**  $ttt = 94 + 19 * \text{length}\ zs$   
**shows** *transforms tm7 tps0 ttt tps7''*  
*<proof>*

**end**

**end**

**lemma** *transforms-tm-numlist-wfI* [*transforms-intros*]:  
**fixes**  $j1\ j2 :: \text{tapeid}\ x$   
**fixes**  $tps\ tps' :: \text{tape list}$  **and**  $zs :: \text{symbol list}$  **and**  $ttt\ k :: \text{nat}$   
**assumes**  $0 < j1\ j1 < k\ j2 + 1 < k\ j1 \neq j2\ 0 < j2\ j1 \neq j2 + 1\ \text{length}\ tps = k$   
**and** *proper-symbols zs*  
**assumes**  
 $tps\ !\ j1 = (\lfloor zs \rfloor, 1)$   
 $tps\ !\ j2 = (\lfloor [] \rfloor, 1)$   
 $tps\ !\ (j2 + 1) = (\lfloor [] \rfloor, 1)$   
**assumes**  $ttt = 94 + 19 * \text{length}\ zs$   
**assumes**  $tps' = tps$   
 $[j2 := (\lfloor \text{numlist-wf}\ zs \rfloor_B, 1)]$   
**shows** *transforms (tm-numlist-wf j1 j2) tps ttt tps'*  
*<proof>*

## 2.12.5 Checking well-formedness for lists of lists

The next Turing machine checks all conditions from the criterion in lemma *numlistlist-wf-iff* one after another for the symbols on tape  $j_1$  and writes to tape  $j_2$  either the number 1 or 0 depending on whether all conditions were met. It assumes tape  $j_2$  is initialized with 0.

**definition** *tm-numlistlist-wf* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-numlistlist-wf j1 j2  $\equiv$ 
  tm-proper-symbols-lt j1 j2 6 ;;
  tm-not-has2 0 | j1 (j2 + 1) ;;
  tm-and j2 (j2 + 1) ;;
  tm-setn (j2 + 1) 0 ;;
  tm-empty-or-endswith j1 (j2 + 1) # ;;
  tm-and j2 (j2 + 1) ;;
  tm-setn (j2 + 1) 0 ;;
  tm-not-has2 0 # j1 (j2 + 1) ;;
  tm-and j2 (j2 + 1) ;;
  tm-setn (j2 + 1) 0 ;;
  tm-not-has2 1 # j1 (j2 + 1) ;;
  tm-and j2 (j2 + 1) ;;
  tm-setn (j2 + 1) 0

```

**lemma** *tm-numlistlist-wf-tm*:

```

assumes  $k \geq 2$  and  $G \geq 6$  and  $0 < j_2 < j_1$  and  $j_1 < k$   $j_2 + 1 < k$ 
shows turing-machine  $k$   $G$  (tm-numlistlist-wf j1 j2)
<proof>

```

**locale** *turing-machine-numlistlist-wf* =

**fixes**  $j_1$   $j_2$  :: *tapeidx*

**begin**

**definition** *tm1*  $\equiv$  *tm-proper-symbols-lt* j1 j2 6

**definition** *tm2*  $\equiv$  *tm1* ;; *tm-not-has2* 0 | j1 (j2 + 1)

**definition** *tm3*  $\equiv$  *tm2* ;; *tm-and* j2 (j2 + 1)

**definition** *tm4*  $\equiv$  *tm3* ;; *tm-setn* (j2 + 1) 0

**definition** *tm5*  $\equiv$  *tm4* ;; *tm-empty-or-endswith* j1 (j2 + 1) #

**definition** *tm6*  $\equiv$  *tm5* ;; *tm-and* j2 (j2 + 1)

**definition** *tm7*  $\equiv$  *tm6* ;; *tm-setn* (j2 + 1) 0

**definition** *tm8*  $\equiv$  *tm7* ;; *tm-not-has2* 0 # j1 (j2 + 1)

**definition** *tm9*  $\equiv$  *tm8* ;; *tm-and* j2 (j2 + 1)

**definition** *tm10*  $\equiv$  *tm9* ;; *tm-setn* (j2 + 1) 0

**definition** *tm11*  $\equiv$  *tm10* ;; *tm-not-has2* 1 # j1 (j2 + 1)

**definition** *tm12*  $\equiv$  *tm11* ;; *tm-and* j2 (j2 + 1)

**definition** *tm13*  $\equiv$  *tm12* ;; *tm-setn* (j2 + 1) 0

**lemma** *tm13-eq-tm-numlistlist-wf*: *tm13* = *tm-numlistlist-wf* j1 j2

<proof>

**context**

**fixes** *tps0* :: *tape list* **and** *zs* :: *symbol list* **and**  $k$  :: *nat*

**assumes** *zs*: *proper-symbols* *zs*

**assumes** *jk*:  $0 < j_1$   $j_1 < k$   $j_2 + 1 < k$   $j_1 \neq j_2$   $0 < j_2$   $j_1 \neq j_2 + 1$  *length* *tps0* =  $k$

**assumes** *tps0*:

$tps0 ! j_1 = (\lfloor zs \rfloor, 1)$

$tps0 ! j_2 = (\lfloor [] \rfloor, 1)$

$tps0 ! (j_2 + 1) = (\lfloor [] \rfloor, 1)$

**begin**

**definition** *tps1*  $\equiv$  *tps0*

$[j_2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \rfloor_B, 1)]$

**lemma** *tm1* [*transforms-intros*]:

**assumes** *ttt* =  $5 + 7 * \text{length } zs$

**shows** *transforms* *tm1* *tps0* *ttt* *tps1*

*<proof>*

**definition**  $tps2 \equiv tps0$

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor \text{if has2 } zs \text{ } \mathbf{0} \mid \text{then } 0 \text{ else } 1 \rfloor_N, 1)]$

**lemma**  $tm2$  [*transforms-intros*]:

**assumes**  $ttt = 46 + 17 * \text{length } zs$

**shows** *transforms*  $tm2$   $tps0$   $ttt$   $tps2$

*<proof>*

**definition**  $tps3 \equiv tps0$

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \text{ } \mathbf{0} \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor \text{if has2 } zs \text{ } \mathbf{0} \mid \text{then } 0 \text{ else } 1 \rfloor_N, 1)]$

**lemma**  $tm3$  [*transforms-intros*]:

**assumes**  $ttt = 46 + 17 * \text{length } zs + 3$

**shows** *transforms*  $tm3$   $tps0$   $ttt$   $tps3$

*<proof>*

**definition**  $tps4 \equiv tps0$

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \text{ } \mathbf{0} \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor 0 \rfloor_N, 1)]$

**lemma**  $tm4$ :

**assumes**  $ttt = 46 + 17 * \text{length } zs + 13 + 2 * \text{nlength (if has2 } zs \text{ } \mathbf{0} \mid \text{then } 0 \text{ else } 1)$

**shows** *transforms*  $tm4$   $tps0$   $ttt$   $tps4$

*<proof>*

**lemma**  $tm4'$  [*transforms-intros*]:

**assumes**  $ttt = 46 + 17 * \text{length } zs + 15$

**shows** *transforms*  $tm4$   $tps0$   $ttt$   $tps4$

*<proof>*

**definition**  $tps5 \equiv tps0$

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \text{ } \mathbf{0} \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor zs = [] \vee \text{last } zs = \# \rfloor_B, 1)]$

**lemma**  $tm5$  [*transforms-intros*]:

**assumes**  $ttt = 79 + 19 * \text{length } zs$

**shows** *transforms*  $tm5$   $tps0$   $ttt$   $tps5$

*<proof>*

**definition**  $tps6 \equiv tps0$

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \text{ } \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor zs = [] \vee \text{last } zs = \# \rfloor_B, 1)]$

**lemma**  $tm6$  [*transforms-intros*]:

**assumes**  $ttt = 82 + 19 * \text{length } zs$

**shows** *transforms*  $tm6$   $tps0$   $ttt$   $tps6$

*<proof>*

**definition**  $tps7 \equiv tps0$

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \text{ } \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor 0 \rfloor_N, 1)]$

**lemma**  $tm7$ :

**assumes**  $ttt = 92 + 19 * \text{length } zs + 2 * \text{nlength (if } zs = [] \vee \text{last } zs = \# \text{ then } 1 \text{ else } 0)$

**shows** *transforms*  $tm7$   $tps0$   $ttt$   $tps7$

*<proof>*

**lemma**  $tm7'$  [*transforms-intros*]:

**assumes**  $ttt = 94 + 19 * \text{length } zs$

**shows** *transforms tm7 tps0 ttt tps7*  
 ⟨*proof*⟩

**definition** *tps8*  $\equiv$  *tps0*

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor \text{if has2 } zs \mathbf{0} \# \text{ then } 0 \text{ else } 1 \rfloor_N, 1)]$

**lemma** *tm8* [*transforms-intros*]:

**assumes**  $ttt = 135 + 29 * \text{length } zs$   
**shows** *transforms tm8 tps0 ttt tps8*  
 ⟨*proof*⟩

**definition** *tps9*  $\equiv$  *tps0*

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \wedge \neg \text{has2 } zs \mathbf{0} \# \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor \text{if has2 } zs \mathbf{0} \# \text{ then } 0 \text{ else } 1 \rfloor_N, 1)]$

**lemma** *tm9* [*transforms-intros*]:

**assumes**  $ttt = 138 + 29 * \text{length } zs$   
**shows** *transforms tm9 tps0 ttt tps9*  
 ⟨*proof*⟩

**definition** *tps10*  $\equiv$  *tps0*

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \wedge \neg \text{has2 } zs \mathbf{0} \# \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor 0 \rfloor_N, 1)]$

**lemma** *tm10*:

**assumes**  $ttt = 148 + 29 * \text{length } zs + 2 * \text{nlength } (\text{if has2 } zs \mathbf{0} \# \text{ then } 0 \text{ else } 1)$   
**shows** *transforms tm10 tps0 ttt tps10*  
 ⟨*proof*⟩

**lemma** *tm10'* [*transforms-intros*]:

**assumes**  $ttt = 150 + 29 * \text{length } zs$   
**shows** *transforms tm10 tps0 ttt tps10*  
 ⟨*proof*⟩

**definition** *tps11*  $\equiv$  *tps0*

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \wedge \neg \text{has2 } zs \mathbf{0} \# \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor \text{if has2 } zs \mathbf{1} \# \text{ then } 0 \text{ else } 1 \rfloor_N, 1)]$

**lemma** *tm11* [*transforms-intros*]:

**assumes**  $ttt = 191 + 39 * \text{length } zs$   
**shows** *transforms tm11 tps0 ttt tps11*  
 ⟨*proof*⟩

**definition** *tps12*  $\equiv$  *tps0*

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \wedge \neg \text{has2 } zs \mathbf{0} \# \wedge \neg \text{has2 } zs \mathbf{1} \# \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor \text{if has2 } zs \mathbf{1} \# \text{ then } 0 \text{ else } 1 \rfloor_N, 1)]$

**lemma** *tm12* [*transforms-intros*]:

**assumes**  $ttt = 194 + 39 * \text{length } zs$   
**shows** *transforms tm12 tps0 ttt tps12*  
 ⟨*proof*⟩

**definition** *tps13*  $\equiv$  *tps0*

$[j2 := (\lfloor \text{proper-symbols-lt } 6 \text{ } zs \wedge \neg \text{has2 } zs \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \wedge \neg \text{has2 } zs \mathbf{0} \# \wedge \neg \text{has2 } zs \mathbf{1} \# \rfloor_B, 1),$   
 $j2 + 1 := (\lfloor 0 \rfloor_N, 1)]$

**lemma** *tm13*:

**assumes**  $ttt = 204 + 39 * \text{length } zs + 2 * \text{nlength } (\text{if has2 } zs \mathbf{1} \# \text{ then } 0 \text{ else } 1)$   
**shows** *transforms tm13 tps0 ttt tps13*  
 ⟨*proof*⟩

**lemma** *tm13'*:

**assumes**  $ttt = 206 + 39 * \text{length } zs$   
**shows** *transforms tm13 tps0 ttt tps13*  
 ⟨*proof*⟩

**definition**  $tps13' \equiv tps0$

$[j2 := (\lfloor \text{proper-symbols-}lt\ 6\ zs \wedge \neg \text{has2 } zs\ \mathbf{0} \mid \wedge (zs = [] \vee \text{last } zs = \#) \wedge \neg \text{has2 } zs\ \mathbf{0}\ \# \wedge \neg \text{has2 } zs\ \mathbf{1}\ \# \rfloor_B, 1)]$

**lemma** *tm13''*:

**assumes**  $ttt = 206 + 39 * \text{length } zs$   
**shows** *transforms tm13 tps0 ttt tps13'*  
 ⟨*proof*⟩

**definition**  $tps13'' \equiv tps0$

$[j2 := (\lfloor \text{numlistlist-wf } zs \rfloor_B, 1)]$

**lemma** *tm13'''*:

**assumes**  $ttt = 206 + 39 * \text{length } zs$   
**shows** *transforms tm13 tps0 ttt tps13''*  
 ⟨*proof*⟩

**end**

**end**

**lemma** *transforms-tm-numlistlist-wfI* [*transforms-intros*]:

**fixes**  $j1\ j2 :: \text{tapeidx}$

**fixes**  $tps\ tps' :: \text{tape list}$  **and**  $zs :: \text{symbol list}$  **and**  $ttt\ k :: \text{nat}$

**assumes**  $0 < j1\ j1 < k\ j2 + 1 < k\ j1 \neq j2\ 0 < j2\ j1 \neq j2 + 1\ \text{length } tps = k$   
**and** *proper-symbols*  $zs$

**assumes**

$tps ! j1 = (\lfloor zs \rfloor, 1)$

$tps ! j2 = (\lfloor [] \rfloor, 1)$

$tps ! (j2 + 1) = (\lfloor [] \rfloor, 1)$

**assumes**  $ttt = 206 + 39 * \text{length } zs$

**assumes**  $tps' = tps$

$[j2 := (\lfloor \text{numlistlist-wf } zs \rfloor_B, 1)]$

**shows** *transforms (tm-numlistlist-wf j1 j2) tps ttt tps'*

⟨*proof*⟩

**end**

# Chapter 3

## Time complexity

```
theory NP
imports Elementary Composing Symbol-Ops
begin
```

In order to formulate the Cook-Levin theorem we need to formalize **SAT** and  $\mathcal{NP}$ -completeness. This chapter is devoted to the latter and hence introduces the complexity class  $\mathcal{NP}$  and the concept of polynomial-time many-one reduction. Moreover, although not required for the Cook-Levin theorem, it introduces the class  $\mathcal{P}$  and contains a proof of  $\mathcal{P} \subseteq \mathcal{NP}$  (see Section 3.3). The chapter concludes with some easy results about  $\mathcal{P}$ ,  $\mathcal{NP}$  and reducibility in Section 3.4.

### 3.1 The time complexity classes **DTIME**, $\mathcal{P}$ , and $\mathcal{NP}$

Arora and Barak [2, Definitions 1.12, 1.13] define  $\text{DTIME}(T(n))$  as the set of all languages that can be decided in time  $c \cdot T(n)$  for some  $c > 0$  and  $\mathcal{P} = \bigcup_{c \geq 1} \text{DTIME}(n^c)$ . However since  $0^c = 0$  for  $c \geq 1$ , this means that for a language  $L$  to be in  $\mathcal{P}$ , the Turing machine deciding  $L$  must check the empty string in zero steps. For a Turing machine to halt in zero steps, it must start in the halting state, which limits its usefulness. Because of this technical issue we define  $\text{DTIME}(T(n))$  as the set of all languages that can be decided with a running time in  $O(T(n))$ , which seems a common enough alternative [11, 12, 1].

Languages are sets of strings, and deciding a language means computing its characteristic function.

```
type-synonym language = string set
```

```
definition characteristic :: language  $\Rightarrow$  (string  $\Rightarrow$  string) where
  characteristic L  $\equiv$  ( $\lambda x$ . [x  $\in$  L])
```

```
definition DTIME :: (nat  $\Rightarrow$  nat)  $\Rightarrow$  language set where
  DTIME T  $\equiv$  {L.  $\exists k G M T'$ .
    turing-machine k G M  $\wedge$ 
    big-oh T' T  $\wedge$ 
    computes-in-time k M (characteristic L) T'}
```

```
definition complexity-class-P :: language set ( $\langle \mathcal{P} \rangle$ ) where
  P  $\equiv$   $\bigcup_{c \in \{1..\}}$ . DTIME ( $\lambda n$ . n  $\wedge$  c)
```

A language  $L$  is in  $\mathcal{NP}$  if there is a polynomial  $p$  and a polynomial-time Turing machine, called the *verifier*, such that for all strings  $x \in \{\mathbf{0}, \mathbf{1}\}^*$ ,

$$x \in L \iff \exists u \in \{\mathbf{0}, \mathbf{1}\}^{p(|x|)} : M(\langle x, u \rangle) = \mathbf{I}.$$

The string  $u$  does not seem to have a name in general, but in case the verifier outputs  $\mathbf{I}$  on input  $\langle x, u \rangle$  it is called a *certificate* for  $x$  [2, Definition 2.1].

```
definition complexity-class-NP :: language set ( $\langle \mathcal{NP} \rangle$ ) where
  NP  $\equiv$  {L.  $\exists k G M p T$  verify.
    turing-machine k G M  $\wedge$ 
    polynomial p  $\wedge$ 
```

*big-oh-poly*  $T \wedge$   
*computes-in-time*  $k M \text{fverify } T \wedge$   
 $(\forall x. x \in L \iff (\exists u. \text{length } u = p(\text{length } x) \wedge \text{fverify } \langle x, u \rangle = [\mathbb{I}]))$

The definition of  $\mathcal{NP}$  is the one place where we need an actual polynomial function, namely  $p$ , rather than a function that is merely bounded by a polynomial. This raises the question as to the definition of a polynomial function. Arora and Barak [2] do not seem to give a definition in the context of  $\mathcal{NP}$  but explicitly state that polynomial functions are mappings  $\mathbb{N} \rightarrow \mathbb{N}$ . Presumably they also have the form  $f(n) = \sum_{i=0}^d a_i \cdot n^i$ , as polynomials do. We have filled in the gap in this definition in Section 2.1.4 by letting the coefficients  $a_i$  be natural numbers.

Regardless of whether this is the meaning intended by Arora and Barak, the choice is justified because with it the verifier-based definition of  $\mathcal{NP}$  is equivalent to the original definition via nondeterministic Turing machines (NTMs). In the usual equivalence proof (for example, Arora and Barak [2, Theorem 2.6]) a verifier TM and an NTM are constructed.

For the one direction, if a language is decided by a polynomial-time NTM then a verifier TM can be constructed that simulates the NTM on input  $x$  by using the bits in the string  $u$  for the nondeterministic choices. The strings  $u$  have the length  $p(|x|)$ . So for this construction to work, there must be a polynomial  $p$  that bounds the running time of the NTM. Clearly, a polynomial function with natural coefficients exists with that property.

For the other direction, assume a language has a verifier TM where  $p$  is a polynomial function with natural coefficients. An NTM deciding this language receives  $x$  as input, then “guesses” a string  $u$  of length  $p(|x|)$ , and then runs the verifier on the pair  $\langle x, u \rangle$ . For this NTM to run in polynomial time,  $p$  must be computable in time polynomial in  $|x|$ . We have shown this to be the case in lemma *transforms-tm-polynomialI* in Section 2.7.8.

A language  $L_1$  is polynomial-time many-one reducible to a language  $L_2$  if there is a polynomial-time computable function  $f_{\text{reduce}}$  such that for all  $x$ ,  $x \in L_1$  iff.  $f_{\text{reduce}}(x) \in L_2$ .

**definition** *reducible* (*infix*  $\langle \leq_p \rangle$  50) **where**

$L_1 \leq_p L_2 \equiv \exists k G M T \text{freduce.}$   
*turing-machine*  $k G M \wedge$   
*big-oh-poly*  $T \wedge$   
*computes-in-time*  $k M \text{freduce } T \wedge$   
 $(\forall x. x \in L_1 \iff \text{freduce } x \in L_2)$

**abbreviation** *NP-hard* :: *language*  $\Rightarrow$  *bool* **where**

*NP-hard*  $L \equiv \forall L' \in \mathcal{NP}. L' \leq_p L$

**definition** *NP-complete* :: *language*  $\Rightarrow$  *bool* **where**

*NP-complete*  $L \equiv L \in \mathcal{NP} \wedge \text{NP-hard } L$

Requiring  $c \geq 1$  in the definition of  $\mathcal{P}$  is not essential:

**lemma** *in-P-iff*:  $L \in \mathcal{P} \iff (\exists c. L \in \text{DTIME } (\lambda n. n \hat{\ } c))$   
*(proof)*

## 3.2 Restricting verifiers to one-bit output

The verifier Turing machine in the definition of  $\mathcal{NP}$  can output any symbol sequence. In this section we restrict it to outputting only the symbol sequence  $\mathbf{1}$  or  $\mathbf{0}$ . This is equivalent to the definition because it is easy to check if a symbol sequence differs from  $\mathbf{1}$  and if so change it to  $\mathbf{0}$ , as we show below.

The advantage of this restriction is that if we can make the TM halt with the output tape head on cell number 1, the output tape symbol read in the final step equals the output of the TM. We will exploit this in Chapter 6, where we show how to reduce any language  $L \in \mathcal{NP}$  to **SAT**.

The next Turing machine checks if the symbol sequence on tape  $j$  differs from the one-symbol sequence  $\mathbf{1}$  and if so turns it into  $\mathbf{0}$ . It thus ensures that the tape contains only one bit symbol.

**definition** *tm-make-bit* :: *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-make-bit*  $j \equiv$   
*tm-cr*  $j$  ;;

```

IF  $\lambda rs. rs ! j = 1$  THEN
  tm-right j ;;
  IF  $\lambda rs. rs ! j = \square$  THEN
    []
  ELSE
    tm-set j [0]
  ENDIF
ELSE
  tm-set j [0]
ENDIF

```

**lemma** *tm-make-bit-tm*:

**assumes**  $G \geq 4$  **and**  $0 < j$  **and**  $j < k$   
**shows** *turing-machine k G (tm-make-bit j)*  
*<proof>*

**locale** *turing-machine-make-bit* =

**fixes**  $j :: \text{tapeidx}$   
**begin**

**definition**  $tm1 \equiv tm-cr\ j$

**definition**  $tmT1 \equiv tm-right\ j$

**definition**  $tmT12 \equiv IF\ \lambda rs. rs ! j = \square\ THEN\ []\ ELSE\ tm-set\ j\ [0]\ ENDIF$

**definition**  $tmT2 \equiv tmT1\ ;;\ tmT12$

**definition**  $tm12 \equiv IF\ \lambda rs. rs ! j = 1\ THEN\ tmT2\ ELSE\ tm-set\ j\ [0]\ ENDIF$

**definition**  $tm2 \equiv tm1\ ;;\ tm12$

**lemma** *tm2-eq-tm-make-bit*:  $tm2 \equiv tm-make-bit\ j$

*<proof>*

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $zs :: \text{symbol list}$

**assumes**  $jk: j < \text{length}\ tps0$

**and**  $zs: \text{proper-symbols}\ zs$

**assumes**  $tps0: tps0 :: j = \lfloor zs \rfloor$

**begin**

**lemma** *clean: clean-tape (tps0 ! j)*

*<proof>*

**definition**  $tps1 \equiv tps0[j := (\lfloor zs \rfloor, 1)]$

**lemma** *tm1 [transforms-intros]*:

**assumes**  $ttt = tps0 : \# : j + 2$

**shows** *transforms tm1 tps0 ttt tps1*

*<proof>*

**definition**  $tpsT1 \equiv tps0[j := (\lfloor zs \rfloor, 2)]$

**lemma** *tmT1 [transforms-intros]*:

**assumes**  $ttt = 1$

**shows** *transforms tmT1 tps1 ttt tpsT1*

*<proof>*

**definition**  $tpsT2 \equiv tps0$

$j := \text{if length } zs \leq 1 \text{ then } (\lfloor zs \rfloor, 2) \text{ else } (\lfloor [0] \rfloor, 1)$

**lemma** *tmT12 [transforms-intros]*:

**assumes**  $ttt = 14 + 2 * \text{length}\ zs$

**shows** *transforms tmT12 tpsT1 ttt tpsT2*

*<proof>*

**lemma** *tmT2 [transforms-intros]*:

```

assumes  $ttt = 15 + 2 * \text{length } zs$ 
shows transforms tmT2 tps1 ttt tpsT2
⟨proof⟩

```

```

definition  $tps2 \equiv tps0$ 
 $[j := \text{if } zs = [\mathbf{1}] \text{ then } (\lfloor zs \rfloor, 2) \text{ else } (\lfloor [\mathbf{0}] \rfloor, 1)]$ 

```

```

lemma tm12 [transforms-intros]:
assumes  $ttt = 17 + 2 * \text{length } zs$ 
shows transforms tm12 tps1 ttt tps2
⟨proof⟩

```

```

lemma tm2:
assumes  $ttt = 19 + 2 * \text{length } zs + tps0 \text{ \#} j$ 
shows transforms tm2 tps0 ttt tps2
⟨proof⟩

```

**end**

**end**

```

lemma transforms-tm-make-bitI [transforms-intros]:
fixes  $j :: \text{tapeidx}$ 
fixes  $tps \ tps' :: \text{tape list}$  and  $zs :: \text{symbol list}$  and  $ttt :: \text{nat}$ 
assumes  $j < \text{length } tps$  and proper-symbols zs
assumes  $tps \text{ \#} j = \lfloor zs \rfloor$ 
assumes  $ttt = 19 + 2 * \text{length } zs + tps \text{ \#} j$ 
assumes  $tps' = tps$ 
 $[j := \text{if } zs = [\mathbf{1}] \text{ then } (\lfloor zs \rfloor, 2) \text{ else } (\lfloor [\mathbf{0}] \rfloor, 1)]$ 
shows transforms (tm-make-bit j) tps ttt tps'
⟨proof⟩

```

```

lemma output-length-le-time:
assumes turing-machine k G M
and  $tps \text{ \#} 1 = \lfloor zs \rfloor$ 
and proper-symbols zs
and transforms M (snd (start-config k xs)) t tps
shows  $\text{length } zs \leq t$ 
⟨proof⟩

```

This is the alternative definition of  $\mathcal{NP}$ , which restricts the verifier to output only strings of length one:

```

lemma NP-output-len-1:
 $\mathcal{NP} = \{L. \exists k \ G \ M \ p \ T \ \text{fverify.}$ 
 $\text{turing-machine } k \ G \ M \wedge$ 
 $\text{polynomial } p \wedge$ 
 $\text{big-oh-poly } T \wedge$ 
 $\text{computes-in-time } k \ M \ \text{fverify } T \wedge$ 
 $(\forall y. \text{length } (\text{fverify } y) = 1) \wedge$ 
 $(\forall x. x \in L \longleftrightarrow (\exists u. \text{length } u = p (\text{length } x) \wedge \text{fverify } \langle x, u \rangle = [\mathbf{1}])))\}$ 
(is - = ?M)
⟨proof⟩

```

### 3.3 $\mathcal{P}$ is a subset of $\mathcal{NP}$

Let  $L \in \mathcal{P}$  be a language and  $M$  a Turing machine that decides  $L$  in polynomial time. To show  $L \in \mathcal{NP}$  we could use a TM that extracts the first element from the input  $\langle x, u \rangle$  and runs  $M$  on  $x$ . We do not have to construct such a TM explicitly because we have shown that the extraction of the first pair element is computable in polynomial time (lemma *tm-first-computes*), and by assumption the characteristic function of  $L$  is computable in polynomial time, too. The composition of these two functions is the verifier required by the definition of  $\mathcal{NP}$ . And by lemma *computes-composed-poly* the composition of polynomial-time functions is polynomial-time, too.

**theorem** *P-subseteq-NP*:  $\mathcal{P} \subseteq \mathcal{NP}$   
(proof)

### 3.4 More about $\mathcal{P}$ , $\mathcal{NP}$ , and reducibility

We prove some low-hanging fruits about the concepts introduced in this chapter. None of the results are needed to show the Cook-Levin theorem.

A language can be reduced to itself by the identity function. Hence reducibility is a reflexive relation.

**lemma** *reducible-refl*:  $L \leq_p L$   
(proof)

Reducibility is also transitive. If  $L_1 \leq_p L_2$  by a TM  $M_1$  and  $L_2 \leq_p L_3$  by a TM  $M_2$  we merely have to run  $M_2$  on the output of  $M_1$  to show that  $L_1 \leq_p L_3$ . Again this is merely the composition of two polynomial-time computable functions.

**lemma** *reducible-trans*:  
**assumes**  $L_1 \leq_p L_2$  **and**  $L_2 \leq_p L_3$   
**shows**  $L_1 \leq_p L_3$   
(proof)

The usual way to show that a language is  $\mathcal{NP}$ -hard is to reduce another  $\mathcal{NP}$ -hard language to it.

**lemma** *ex-reducible-imp-NP-hard*:  
**assumes** *NP-hard*  $L'$  **and**  $L' \leq_p L$   
**shows** *NP-hard*  $L$   
(proof)

The converse is also true because reducibility is a reflexive relation.

**lemma** *NP-hard-iff-reducible*: *NP-hard*  $L \iff (\exists L'. \text{NP-hard } L' \wedge L' \leq_p L)$   
(proof)

**lemma** *NP-complete-reducible*:  
**assumes** *NP-complete*  $L'$  **and**  $L \in \mathcal{NP}$  **and**  $L' \leq_p L$   
**shows** *NP-complete*  $L$   
(proof)

In a sense the complexity class  $\mathcal{P}$  is closed under reduction.

**lemma** *P-closed-reduction*:  
**assumes**  $L \in \mathcal{P}$  **and**  $L' \leq_p L$   
**shows**  $L' \in \mathcal{P}$   
(proof)

The next lemmas are items 2 and 3 of Theorem 2.8 of the textbook [2]. Item 1 is the transitivity of the reduction, already proved in lemma *reducible-trans*.

**lemma** *P-eq-NP*:  
**assumes** *NP-hard*  $L$  **and**  $L \in \mathcal{P}$   
**shows**  $\mathcal{P} = \mathcal{NP}$   
(proof)

**lemma** *NP-complete-imp*:  
**assumes** *NP-complete*  $L$   
**shows**  $L \in \mathcal{P} \iff \mathcal{P} = \mathcal{NP}$   
(proof)

**end**

# Chapter 4

## Satisfiability

```
theory Satisfiability
  imports Wellformed NP
begin
```

This chapter introduces the language **SAT** and shows that it is in  $\mathcal{NP}$ , which constitutes the easier part of the Cook-Levin theorem. The other part, the  $\mathcal{NP}$ -hardness of **SAT**, is what all the following chapters are concerned with.

We first introduce Boolean formulas in conjunctive normal form and the concept of satisfiability. Then we define a way to represent such formulas as bit strings, leading to the definition of the language **SAT** as a set of strings (Section 4.1).

For the proof that **SAT** is in  $\mathcal{NP}$ , we construct a Turing machine that, given a CNF formula and a string representing a variable assignment, outputs **1** iff. the assignment satisfies the formula. The TM will run in polynomial time, and there are always assignments polynomial (in fact, linear) in the length of the formula (Section 4.2).

### 4.1 The language **SAT**

**SAT** is the language of all strings representing satisfiable Boolean formulas in conjunctive normal form (CNF). This section introduces a minimal version of Boolean formulas in conjunctive normal form, including the concepts of assignments and satisfiability.

#### 4.1.1 CNF formulas and satisfiability

Arora and Barak [2, p. 44] define Boolean formulas in general as expressions over  $\wedge, \vee, \neg$ , parentheses, and variables  $v_1, v_2, \dots$  in the usual way. Boolean formulas in conjunctive normal form are defined as  $\bigwedge_i \left( \bigvee_j v_{i_j} \right)$ , where the  $v_{i_j}$  are literals. This definition does not seem to allow for empty clauses. Also whether the “empty CNF formula” exists is somewhat doubtful. Nevertheless, our formalization allows for both empty clauses and the empty CNF formula, because this enables us to represent CNF formulas as lists of clauses and clauses as lists of literals without having to somehow forbid the empty list. This seems to be a popular approach for formalizing CNF formulas in the context of **SAT** and  $\mathcal{NP}$  [7, 14].

We identify a variable  $v_i$  with its index  $i$ , which can be any natural number. A *literal* can either be positive or negative, representing a variable or negated variable, respectively.

```
datatype literal = Neg nat | Pos nat
```

```
type-synonym clause = literal list
```

```
type-synonym formula = clause list
```

An *assignment* maps all variables, given by their index, to a Boolean:

```
type-synonym assignment = nat  $\Rightarrow$  bool
```

**abbreviation** *satisfies-literal* :: *assignment*  $\Rightarrow$  *literal*  $\Rightarrow$  *bool* **where**  
*satisfies-literal*  $\alpha$   $x \equiv$  *case*  $x$  of *Neg*  $n \Rightarrow \neg \alpha$   $n$  | *Pos*  $n \Rightarrow \alpha$   $n$

**definition** *satisfies-clause* :: *assignment*  $\Rightarrow$  *clause*  $\Rightarrow$  *bool* **where**  
*satisfies-clause*  $\alpha$   $c \equiv \exists x \in \text{set } c. \text{ satisfies-literal } \alpha$   $x$

**definition** *satisfies* :: *assignment*  $\Rightarrow$  *formula*  $\Rightarrow$  *bool* (**infix**  $\langle \models \rangle$  60) **where**  
 $\alpha \models \varphi \equiv \forall c \in \text{set } \varphi. \text{ satisfies-clause } \alpha$   $c$

As is customary, the empty clause is satisfied by no assignment, and the empty CNF formula is satisfied by every assignment.

**proposition**  $\neg \text{ satisfies-clause } \alpha$  []  
 $\langle \text{proof} \rangle$

**proposition**  $\alpha \models []$   
 $\langle \text{proof} \rangle$

**lemma** *satisfies-clause-take*:  
**assumes**  $i < \text{length } \text{clause}$   
**shows** *satisfies-clause*  $\alpha$  (*take* (*Suc*  $i$ ) *clause*)  $\longleftrightarrow$   
*satisfies-clause*  $\alpha$  (*take*  $i$  *clause*)  $\vee$  *satisfies-literal*  $\alpha$  (*clause* !  $i$ )  
 $\langle \text{proof} \rangle$

**lemma** *satisfies-take*:  
**assumes**  $i < \text{length } \varphi$   
**shows**  $\alpha \models \text{take } (\text{Suc } i) \varphi \longleftrightarrow \alpha \models \text{take } i \varphi \wedge \text{ satisfies-clause } \alpha$  ( $\varphi$  !  $i$ )  
 $\langle \text{proof} \rangle$

**lemma** *satisfies-append*:  
**assumes**  $\alpha \models \varphi_1 @ \varphi_2$   
**shows**  $\alpha \models \varphi_1$  **and**  $\alpha \models \varphi_2$   
 $\langle \text{proof} \rangle$

**lemma** *satisfies-append'*:  
**assumes**  $\alpha \models \varphi_1$  **and**  $\alpha \models \varphi_2$   
**shows**  $\alpha \models \varphi_1 @ \varphi_2$   
 $\langle \text{proof} \rangle$

**lemma** *satisfies-concat-map*:  
**assumes**  $\alpha \models \text{concat } (\text{map } f [0..<k])$  **and**  $i < k$   
**shows**  $\alpha \models f$   $i$   
 $\langle \text{proof} \rangle$

**lemma** *satisfies-concat-map'*:  
**assumes**  $\bigwedge i. i < k \implies \alpha \models f$   $i$   
**shows**  $\alpha \models \text{concat } (\text{map } f [0..<k])$   
 $\langle \text{proof} \rangle$

The main ingredient for defining SAT is the concept of *satisfiable* CNF formula:

**definition** *satisfiable* :: *formula*  $\Rightarrow$  *bool* **where**  
*satisfiable*  $\varphi \equiv \exists \alpha. \alpha \models \varphi$

The set of all variables used in a CNF formula is finite.

**definition** *variables* :: *formula*  $\Rightarrow$  *nat set* **where**  
*variables*  $\varphi \equiv \{n. \exists c \in \text{set } \varphi. \text{ Neg } n \in \text{set } c \vee \text{ Pos } n \in \text{set } c\}$

**lemma** *finite-variables*: *finite* (*variables*  $\varphi$ )  
 $\langle \text{proof} \rangle$

**lemma** *variables-append*: *variables* ( $\varphi_1 @ \varphi_2$ ) = *variables*  $\varphi_1 \cup$  *variables*  $\varphi_2$   
 $\langle \text{proof} \rangle$

Arora and Barak [2, Claim 2.13] define the *size* of a CNF formula as the number of  $\wedge/\vee$  symbols. We use a slightly different definition, namely the number of literals:

**definition**  $fsize :: formula \Rightarrow nat$  **where**  
 $fsize \varphi \equiv sum-list (map length \varphi)$

## 4.1.2 Predicates on assignments

Every CNF formula is satisfied by a set of assignments. Conversely, for certain sets of assignments we can construct CNF formulas satisfied by exactly these assignments. This will be helpful later when we construct formulas for reducing arbitrary languages to SAT (see Section 6).

### Universality of CNF formulas

A set (represented by a predicate)  $F$  of assignments depends on the first  $\ell$  variables iff. any two assignments that agree on the first  $\ell$  variables are either both in the set or both outside of the set.

**definition**  $depon :: nat \Rightarrow (assignment \Rightarrow bool) \Rightarrow bool$  **where**  
 $depon \ell F \equiv \forall \alpha_1 \alpha_2. (\forall i < \ell. \alpha_1 i = \alpha_2 i) \longrightarrow F \alpha_1 = F \alpha_2$

Lists of all strings of the same length:

**fun**  $str-of-len :: nat \Rightarrow string list$  **where**  
 $str-of-len 0 = [] \mid$   
 $str-of-len (Suc \ell) = map ((\#) \mathbf{0}) (str-of-len \ell) @ map ((\#) \mathbf{1}) (str-of-len \ell)$

**lemma**  $length-str-of-len: length (str-of-len \ell) = 2^\ell$   
 $\langle proof \rangle$

**lemma**  $in-str-of-len-length: xs \in set (str-of-len \ell) \Longrightarrow length xs = \ell$   
 $\langle proof \rangle$

**lemma**  $length-in-str-of-len: length xs = \ell \Longrightarrow xs \in set (str-of-len \ell)$   
 $\langle proof \rangle$

A predicate  $F$  depending on the first  $\ell$  variables  $v_0, \dots, v_{\ell-1}$  can be regarded as a truth table over  $\ell$  variables. The next lemma shows that for every such truth table there exists a CNF formula with at most  $2^\ell$  clauses and  $\ell \cdot 2^\ell$  literals over the first  $\ell$  variables. This is the well-known fact that every Boolean function (over  $\ell$  variables) can be represented by a CNF formula [2, Claim 2.13].

**lemma**  $depon-ex-formula:$

**assumes**  $depon \ell F$

**shows**  $\exists \varphi.$

$fsize \varphi \leq \ell * 2^\ell \wedge$

$length \varphi \leq 2^\ell \wedge$

$variables \varphi \subseteq \{..<\ell\} \wedge$

$(\forall \alpha. F \alpha = \alpha \models \varphi)$

$\langle proof \rangle$

### Substitutions of variables

We will sometimes consider CNF formulas over the first  $\ell$  variables and derive other CNF formulas from them by substituting these variables. Such a substitution will be represented by a list  $\sigma$  of length at least  $\ell$ , meaning that the variable  $v_i$  is replaced by  $v_{\sigma(i)}$ . We will call this operation on formulas  $relabel$ , and the corresponding one on literals  $rename$ :

**fun**  $rename :: nat list \Rightarrow literal \Rightarrow literal$  **where**

$rename \sigma (Neg i) = Neg (\sigma ! i) \mid$

$rename \sigma (Pos i) = Pos (\sigma ! i)$

**definition**  $relabel :: nat list \Rightarrow formula \Rightarrow formula$  **where**

$relabel \sigma \varphi \equiv map (map (rename \sigma)) \varphi$

**lemma**  $fsize-relabel: fsize (relabel \sigma \varphi) = fsize \varphi$

$\langle proof \rangle$

A substitution  $\sigma$  can also be applied to an assignment and to a list of variable indices:

**definition** *remap* :: *nat list*  $\Rightarrow$  *assignment*  $\Rightarrow$  *assignment* **where**  
*remap*  $\sigma$   $\alpha$   $i \equiv$  if  $i < \text{length } \sigma$  then  $\alpha (\sigma ! i)$  else  $\alpha i$

**definition** *reseq* :: *nat list*  $\Rightarrow$  *nat list*  $\Rightarrow$  *nat list* **where**  
*reseq*  $\sigma$   $vs \equiv$  map  $((!) \sigma)$   $vs$

**lemma** *length-reseq* [*simp*]: *length* (*reseq*  $\sigma$   $vs$ ) = *length*  $vs$   
*<proof>*

Relabeling a formula and remapping an assignment are equivalent in a sense.

**lemma** *satisfies-sigma*:  
**assumes** *variables*  $\varphi \subseteq \{..<\text{length } \sigma\}$   
**shows**  $\alpha \models \text{relabel } \sigma \varphi \iff \text{remap } \sigma \alpha \models \varphi$   
*<proof>*

### 4.1.3 Representing CNF formulas as strings

By identifying negated literals with even numbers and positive literals with odd numbers, we can identify literals with natural numbers. This yields a straightforward representation of a clause as a list of numbers and of a CNF formula as a list of lists of numbers. Such a list can, in turn, be represented as a symbol sequence over a quaternary alphabet as described in Section 2.9, which ultimately can be encoded over a binary alphabet (see Section 2.10). This is essentially how we represent CNF formulas as strings.

We have to introduce a bunch of functions for mapping between these representations.

**fun** *literal-n* :: *literal*  $\Rightarrow$  *nat* **where**  
*literal-n* (*Neg*  $i$ ) =  $2 * i$  |  
*literal-n* (*Pos*  $i$ ) = *Suc* ( $2 * i$ )

**definition** *n-literal* :: *nat*  $\Rightarrow$  *literal* **where**  
*n-literal*  $n \equiv$  if even  $n$  then *Neg* ( $n \text{ div } 2$ ) else *Pos* ( $n \text{ div } 2$ )

**lemma** *n-literal-n*: *n-literal* (*literal-n*  $x$ ) =  $x$   
*<proof>*

**lemma** *literal-n-literal*: *literal-n* (*n-literal*  $n$ ) =  $n$   
*<proof>*

**definition** *clause-n* :: *clause*  $\Rightarrow$  *nat list* **where**  
*clause-n*  $cl \equiv$  map *literal-n*  $cl$

**definition** *n-clause* :: *nat list*  $\Rightarrow$  *clause* **where**  
*n-clause*  $ns \equiv$  map *n-literal*  $ns$

**lemma** *n-clause-n*: *n-clause* (*clause-n*  $cl$ ) =  $cl$   
*<proof>*

**lemma** *clause-n-clause*: *clause-n* (*n-clause*  $n$ ) =  $n$   
*<proof>*

**definition** *formula-n* :: *formula*  $\Rightarrow$  *nat list list* **where**  
*formula-n*  $\varphi \equiv$  map *clause-n*  $\varphi$

**definition** *n-formula* :: *nat list list*  $\Rightarrow$  *formula* **where**  
*n-formula*  $nss \equiv$  map *n-clause*  $nss$

**lemma** *n-formula-n*: *n-formula* (*formula-n*  $\varphi$ ) =  $\varphi$   
*<proof>*

**lemma** *formula-n-formula*: *formula-n* (*n-formula*  $nss$ ) =  $nss$   
*<proof>*

**definition** *formula-zs* :: *formula*  $\Rightarrow$  *symbol list* **where**

*formula-zs*  $\varphi \equiv \text{numlistlist } (\text{formula-n } \varphi)$

The mapping between formulas and well-formed symbol sequences for lists of lists of numbers is bijective.

**lemma** *formula-n-inj*: *formula-n*  $\varphi_1 = \text{formula-n } \varphi_2 \implies \varphi_1 = \varphi_2$   
 ⟨proof⟩

**definition** *zs-formula* :: *symbol list*  $\Rightarrow$  *formula* **where**  
*zs-formula*  $zs \equiv \text{THE } \varphi. \text{formula-zs } \varphi = zs$

**lemma** *zs-formula*:  
**assumes** *numlistlist-wf*  $zs$   
**shows**  $\exists! \varphi. \text{formula-zs } \varphi = zs$   
 ⟨proof⟩

**lemma** *zs-formula-zs*: *zs-formula* (*formula-zs*  $\varphi$ ) =  $\varphi$   
 ⟨proof⟩

**lemma** *formula-zs-formula*:  
**assumes** *numlistlist-wf*  $zs$   
**shows** *formula-zs* (*zs-formula*  $zs$ ) =  $zs$   
 ⟨proof⟩

There will of course be Turing machines that perform computations on formulas. In order to bound their running time, we need bounds for the length of the symbol representation of formulas.

**lemma** *nlength-literal-n-Pos*: *nlength* (*literal-n* (*Pos*  $n$ ))  $\leq \text{Suc } (\text{nlength } n)$   
 ⟨proof⟩

**lemma** *nlength-literal-n-Neg*: *nlength* (*literal-n* (*Neg*  $n$ ))  $\leq \text{Suc } (\text{nlength } n)$   
 ⟨proof⟩

**lemma** *nllength-formula-n*:  
**fixes**  $V :: \text{nat}$  **and**  $\varphi :: \text{formula}$   
**assumes**  $\bigwedge v. v \in \text{variables } \varphi \implies v \leq V$   
**shows** *nllength* (*formula-n*  $\varphi$ )  $\leq \text{fsize } \varphi * \text{Suc } (\text{Suc } (\text{nlength } V)) + \text{length } \varphi$   
 ⟨proof⟩

Since **SAT** is supposed to be a set of strings rather than symbol sequences, we need to map symbol sequences representing formulas to strings:

**abbreviation** *formula-to-string* :: *formula*  $\Rightarrow$  *string* **where**  
*formula-to-string*  $\varphi \equiv \text{symbols-to-string } (\text{binencode } (\text{numlistlist } (\text{formula-n } \varphi)))$

**lemma** *formula-to-string-inj*:  
**assumes** *formula-to-string*  $\varphi_1 = \text{formula-to-string } \varphi_2$   
**shows**  $\varphi_1 = \varphi_2$   
 ⟨proof⟩

While *formula-to-string* maps every CNF formula to a string, not every string represents a CNF formula. We could just ignore such invalid strings and define **SAT** to only contain well-formed strings. But this would implicitly place these invalid strings in the complement of **SAT**. While this does not cause us any issues here, it would if we were to introduce  $\text{co-}\mathcal{NP}$  and wanted to show that  $\overline{\text{SAT}}$  is in  $\text{co-}\mathcal{NP}$ , as we would then have to deal with the invalid strings. So it feels a little like cheating to ignore the invalid strings like this.

Arora and Barak [2, p. 45 footnote 3] recommend mapping invalid strings to “some fixed formula”. A natural candidate for this fixed formula is the empty CNF, since an invalid string in a sense represents nothing, and the empty CNF formula is represented by the empty string. Since the empty CNF formula is satisfiable this implies that all invalid strings become elements of **SAT**.

We end up with the following definition of the protagonist of this article:

**definition** *SAT* :: *language* **where**  
*SAT*  $\equiv \{\text{formula-to-string } \varphi \mid \varphi. \text{satisfiable } \varphi\} \cup \{x. \neg (\exists \varphi. x = \text{formula-to-string } \varphi)\}$

## 4.2 SAT is in $\mathcal{NP}$

In order to show that SAT is in  $\mathcal{NP}$ , we will construct a polynomial-time Turing machine  $M$  and specify a polynomial function  $p$  such that for every  $x$ ,  $x \in \text{SAT}$  iff. there is a  $u \in \{\mathbf{0}, \mathbf{1}\}^{p(|x|)}$  such that  $M$  outputs  $\mathbf{1}$  on  $\langle x; u \rangle$ .

The idea is straightforward: Let  $\varphi$  be the formula represented by the string  $x$ . Interpret the string  $u$  as a list of variables and interpret this as the assignment that assigns True to only the variables in the list. Then check if the assignment satisfies the formula. This check is “obviously” possible in polynomial time because  $M$  simply has to iterate over all clauses and check if at least one literal per clause is true under the assignment. Checking if a literal is true is simply a matter of checking whether the literal’s variable is in the list  $u$ . If the assignment satisfies  $\varphi$ , output  $\mathbf{1}$ , otherwise the empty symbol sequence.

If  $\varphi$  is unsatisfiable then no assignment, hence no  $u$  no matter the length will make  $M$  output  $\mathbf{1}$ . On the other hand, if  $\varphi$  is satisfiable then there will be a satisfying assignment where a subset of the variables in  $\varphi$  are assigned True. Hence there will be a list of variables of at most roughly the length of  $\varphi$ . So setting the polynomial  $p$  to something like  $p(n) = n$  should suffice.

In fact, as we shall see,  $p(n) = n$  suffices. This is so because in our representation, the string  $x$ , being a list of lists, has slightly more overhead per number than the plain list  $u$  has. Therefore listing all variables in  $\varphi$  is guaranteed to need fewer symbols than  $x$  has.

There are several technical details to work out. First of all, the input to  $M$  need not be a well-formed pair. And if it is, the pair  $\langle x, u \rangle$  has to be decoded into separate components  $x$  and  $u$ . These have to be decoded again from the binary to the quaternary alphabet, which is only possible if both  $x$  and  $u$  comprise only bit symbols ( $\mathbf{01}$ ). Then  $M$  needs to check if the decoded  $x$  and  $u$  are valid symbol sequences for lists (of lists) of numbers. In the case of  $u$  this is particularly finicky because the definition of  $\mathcal{NP}$  requires us to provide a string  $u$  of exactly the length  $p(|x|)$  and so we have to pad  $u$  with extra symbols, which have to be stripped again before the validation can take place.

In the first subsection we describe what the verifier TM has to do in terms of symbol sequences. In the subsections after that we devise a Turing machine that implements this behavior.

### 4.2.1 Verifying SAT instances

Our verifier Turing machine for SAT will implement the following function; that is, on input  $zs$  it will output  $verify\text{-}sat\ zs$ . It performs a number of decodings and well-formedness checks and outputs either  $\mathbf{1}$  or the empty symbol sequence.

**definition**  $verify\text{-}sat :: symbol\ list \Rightarrow symbol\ list\ \mathbf{where}$

```

verify-sat zs  $\equiv$ 
  let
    ys = bindecode zs;
    xs = bindecode (first ys);
    vs = rstrip  $\#$  (bindecode (second ys))
  in
    if even (length (first ys))  $\wedge$  bit-symbols (first ys)  $\wedge$  numlistlist-wf xs
    then if bit-symbols (second ys)  $\wedge$  numlist-wf vs
         then if  $(\lambda v. v \in set\ (zs\text{-}numlist\ vs)) \models zs\text{-}formula\ xs$  then  $\mathbf{[1]}$  else  $\square$ 
         else  $\square$ 
    else  $\mathbf{[1]}$ 

```

Next we show that  $verify\text{-}sat$  behaves as is required of a verifier TM for SAT. Its polynomial running time is the subject of the next subsection.

First we consider the case that  $zs$  encodes a pair  $\langle x, u \rangle$  of strings where  $x$  does not represent a CNF formula. Such an  $x$  is in SAT, hence the verifier TM is supposed to output  $\mathbf{1}$ .

**lemma**  $ex\text{-}phi\text{-}x$ :

```

assumes xs = string-to-symbols x
assumes even (length xs) and numlistlist-wf (bindecode xs)
shows  $\exists \varphi. x = formula\text{-}to\text{-}string\ \varphi$ 
<proof>

```

**lemma**  $verify\text{-}sat\text{-}not\text{-}wf\text{-}phi$ :

```

assumes zs =  $\langle x; u \rangle$  and  $\neg (\exists \varphi. x = formula\text{-}to\text{-}string\ \varphi)$ 

```

**shows** *verify-sat*  $zs = [1]$   
 ⟨*proof*⟩

The next case is that  $zs$  represents a pair  $\langle x, u \rangle$  where  $x$  represents an unsatisfiable CNF formula. This  $x$  is thus not in SAT and the verifier TM must output something different from **1**, such as the empty string, regardless of  $u$ .

**lemma** *verify-sat-not-sat*:  
**fixes**  $\varphi :: \text{formula}$   
**assumes**  $zs = \langle \text{formula-to-string } \varphi; u \rangle$  **and**  $\neg \text{satisfiable } \varphi$   
**shows** *verify-sat*  $zs = []$   
 ⟨*proof*⟩

Next we consider the case in which  $zs$  represents a pair  $\langle x, u \rangle$  where  $x$  represents a satisfiable CNF formula and  $u$  a list of numbers padded at the right with  $\#$  symbols. This  $u$  thus represents a variable assignment, namely the one assigning True to exactly the variables in the list. The verifier TM is to output **1** iff. this assignment satisfies the CNF formula represented by  $x$ .

First we show that stripping away  $\#$  symbols does not damage a symbol sequence representing a list of numbers.

**lemma** *rstrip-numlist-append*:  $\text{rstrip } \# (\text{numlist } vars @ \text{replicate } pad \#) = \text{numlist } vars$   
 (**is**  $\text{rstrip } \# ?zs = ?ys$ )  
 ⟨*proof*⟩

**lemma** *verify-sat-wf*:  
**fixes**  $\varphi :: \text{formula}$  **and**  $pad :: \text{nat}$   
**assumes**  $zs = \langle \text{formula-to-string } \varphi; \text{symbols-to-string } (\text{binencode } (\text{numlist } vars @ \text{replicate } pad \#)) \rangle$   
**shows** *verify-sat*  $zs = (\text{if } (\lambda v. v \in \text{set } vars) \models \varphi \text{ then } [1] \text{ else } [])$   
 ⟨*proof*⟩

Finally we show that for every string  $x$  representing a satisfiable CNF formula there is a list of numbers representing a satisfying assignment and represented by a string of length at most  $|x|$ . That means there is always a string of exactly the length of  $x$  consisting of a satisfying assignment plus some padding symbols.

**lemma** *nlength-remove1*:  
**assumes**  $n \in \text{set } ns$   
**shows**  $\text{nlength } (n \# \text{remove1 } n \ ns) = \text{nlength } ns$   
 ⟨*proof*⟩

**lemma** *nlength-distinct-le*:  
**assumes**  $\text{distinct } ns$   
**and**  $\text{set } ns \subseteq \text{set } ms$   
**shows**  $\text{nlength } ns \leq \text{nlength } ms$   
 ⟨*proof*⟩

**lemma** *nlength-nlength-concat*:  $\text{nlength } nss = \text{nlength } (\text{concat } nss) + \text{length } nss$   
 ⟨*proof*⟩

**fun** *variable* ::  $\text{literal} \Rightarrow \text{nat}$  **where**  
 $\text{variable } (\text{Neg } i) = i \mid$   
 $\text{variable } (\text{Pos } i) = i$

**lemma** *sum-list-eq*:  $ns = ms \implies \text{sum-list } ns = \text{sum-list } ms$   
 ⟨*proof*⟩

**lemma** *nlength-clause-le*:  $\text{nlength } (\text{clause-n } cl) \geq \text{nlength } (\text{map } \text{variable } cl)$   
 ⟨*proof*⟩

**lemma** *nlength-formula-ge*:  $\text{nlength } (\text{formula-n } \varphi) \geq \text{nlength } (\text{map } (\text{map } \text{variable}) \varphi)$   
 ⟨*proof*⟩

**lemma** *variables-shorter-formula*:  
**fixes**  $\varphi :: \text{formula}$  **and**  $vars :: \text{nat list}$   
**assumes**  $\text{set } vars \subseteq \text{variables } \varphi$  **and**  $\text{distinct } vars$

**shows**  $nlength\ vars \leq nllength\ (formula-n\ \varphi)$   
 ⟨proof⟩

**lemma** *ex-assignment-linear-length*:

**assumes** *satisfiable*  $\varphi$

**shows**  $\exists vars. (\lambda v. v \in set\ vars) \models \varphi \wedge nlength\ vars \leq nllength\ (formula-n\ \varphi)$   
 ⟨proof⟩

**lemma** *ex-witness-linear-length*:

**fixes**  $\varphi :: formula$

**assumes** *satisfiable*  $\varphi$

**shows**  $\exists us.$

*bit-symbols*  $us \wedge$

$length\ us = length\ (formula-to-string\ \varphi) \wedge$

$verify-sat\ \langle formula-to-string\ \varphi; symbols-to-string\ us \rangle = [1]$

⟨proof⟩

**lemma** *bit-symbols-verify-sat*: *bit-symbols* (*verify-sat*  $zs$ )

⟨proof⟩

## 4.2.2 A Turing machine for verifying formulas

The core of the function *verify-sat* is the expression  $(\lambda v. v \in set\ (zs-numlist\ vs)) \models zs-formula\ xs$ , which checks if an assignment represented by a list of variable indices satisfies a CNF formula represented by a list of lists of literals. In this section we devise a Turing machine performing this check.

Recall that the numbers 0 and 1 are represented by the empty symbol sequence and the symbol sequence **1**, respectively. The Turing machines in this section are described in terms of numbers.

We start with a Turing machine that checks a clause. The TM accepts on tape  $j_1$  a list of numbers representing an assignment  $\alpha$  and on tape  $j_2$  a list of numbers representing a clause. It outputs on tape  $j_3$  the number 1 if  $\alpha$  satisfies the clause, and otherwise 0. To do this the TM iterates over all literals in the clause and determines the underlying variable and the sign of the literal. If the literal is positive and the variable is in the list representing  $\alpha$  or if the literal is negative and the variable is not in the list, the number 1 is written to the tape  $j_3$ . Otherwise the tape remains unchanged. We assume  $j_3$  is initialized with 0, and so it will be 1 if and only if at least one literal is satisfied by  $\alpha$ .

The TM requires five auxiliary tapes  $j_3 + 1, \dots, j_3 + 5$ . Tape  $j_3 + 1$  stores the literals one at a time, and later the variable; tape  $j_3 + 2$  stores the sign of the literal; tape  $j_3 + 3$  stores whether the variable is contained in  $\alpha$ ; tapes  $j_3 + 4$  and  $j_3 + 5$  are the auxiliary tapes of *tm-contains*.

**definition** *tm-sat-clause* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-sat-clause*  $j_1\ j_2\ j_3 \equiv$

*WHILE*  $\square$  ;  $\lambda rs. rs ! j_2 \neq \square$  *DO*

*tm-nextract* 4  $j_2\ (j_3 + 1)$  ;;

*tm-mod2*  $(j_3 + 1)\ (j_3 + 2)$  ;;

*tm-div2*  $(j_3 + 1)$  ;;

*tm-contains*  $j_1\ (j_3 + 1)\ (j_3 + 3)$  ;;

*IF*  $\lambda rs. rs ! (j_3 + 3) = \square \wedge rs ! (j_3 + 2) = \square \vee rs ! (j_3 + 3) \neq \square \wedge rs ! (j_3 + 2) \neq \square$  *THEN*

*tm-setn*  $j_3\ 1$

*ELSE*

$\square$

*ENDIF* ;;

*tm-setn*  $(j_3 + 1)\ 0$  ;;

*tm-setn*  $(j_3 + 2)\ 0$  ;;

*tm-setn*  $(j_3 + 3)\ 0$

*DONE* ;;

*tm-cr*  $j_2$

**lemma** *tm-sat-clause-tm*:

**assumes**  $k \geq 2$  **and**  $G \geq 5$  **and**  $j_3 + 5 < k$   $0 < j_1$   $j_1 < k$   $j_2 < k$   $j_1 < j_3$

**shows** *turing-machine*  $k\ G\ (tm-sat-clause\ j_1\ j_2\ j_3)$

⟨proof⟩

**locale** *turing-machine-sat-clause* =

**fixes**  $j1\ j2\ j3 :: \text{tapeidx}$   
**begin**

**definition**  $tmL1 \equiv tm\text{-nextract } 4\ j2\ (j3 + 1)$

**definition**  $tmL2 \equiv tmL1 ;; tm\text{-mod2 } (j3 + 1)\ (j3 + 2)$

**definition**  $tmL3 \equiv tmL2 ;; tm\text{-div2 } (j3 + 1)$

**definition**  $tmL4 \equiv tmL3 ;; tm\text{-contains } j1\ (j3 + 1)\ (j3 + 3)$

**definition**  $tmI \equiv IF\ \lambda rs. rs!\ (j3 + 3) = \square \wedge rs!\ (j3 + 2) = \square \vee rs!\ (j3 + 3) \neq \square \wedge rs!\ (j3 + 2) \neq \square$   
*THEN*  $tm\text{-setn } j3\ 1$  *ELSE*  $\square$  *ENDIF*

**definition**  $tmL5 \equiv tmL4 ;; tmI$

**definition**  $tmL6 \equiv tmL5 ;; tm\text{-setn } (j3 + 1)\ 0$

**definition**  $tmL7 \equiv tmL6 ;; tm\text{-setn } (j3 + 2)\ 0$

**definition**  $tmL8 \equiv tmL7 ;; tm\text{-setn } (j3 + 3)\ 0$

**definition**  $tmL \equiv WHILE\ \square ; \lambda rs. rs!\ j2 \neq \square DO\ tmL8\ DONE$

**definition**  $tm2 \equiv tmL ;; tm\text{-cr } j2$

**lemma**  $tm2\text{-eq-}tm\text{-sat-clause}: tm2 = tm\text{-sat-clause } j1\ j2\ j3$   
*<proof>*

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $k :: \text{nat}$  **and**  $\text{vars} :: \text{nat list}$  **and**  $\text{clause} :: \text{clause}$

**assumes**  $jk: 0 < j1\ j1 \neq j2\ j3 + 5 < k\ j1 < j3\ j2 < j3\ 0 < j2\ \text{length } tps0 = k$

**assumes**  $tps0:$

$tps0!\ j1 = nltape'\ \text{vars } 0$

$tps0!\ j2 = nltape'\ (\text{clause-n clause})\ 0$

$tps0!\ j3 = (\lfloor 0 \rfloor_N, 1)$

$tps0!\ (j3 + 1) = (\lfloor 0 \rfloor_N, 1)$

$tps0!\ (j3 + 2) = (\lfloor 0 \rfloor_N, 1)$

$tps0!\ (j3 + 3) = (\lfloor 0 \rfloor_N, 1)$

$tps0!\ (j3 + 4) = (\lfloor 0 \rfloor_N, 1)$

$tps0!\ (j3 + 5) = (\lfloor 0 \rfloor_N, 1)$

**begin**

**abbreviation**  $\text{sat-take } t \equiv \text{satisfies-clause } (\lambda v. v \in \text{set vars})\ (\text{take } t\ \text{clause})$

**definition**  $tpsL :: \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsL\ t \equiv tps0$

$[j2 := nltape'\ (\text{clause-n clause})\ t,$

$j3 := (\lfloor \text{sat-take } t \rfloor_B, 1)]$

**lemma**  $tpsL0: tpsL\ 0 = tps0$   
*<proof>*

**definition**  $tpsL1 :: \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsL1\ t \equiv tps0$

$[j2 := nltape'\ (\text{clause-n clause})\ (\text{Suc } t),$

$j3 := (\lfloor \text{sat-take } t \rfloor_B, 1),$

$j3 + 1 := (\lfloor \text{literal-n } (\text{clause}!\ t) \rfloor_N, 1)]$

**lemma**  $tmL1$  [*transforms-intros*]:

**assumes**  $ttt = 12 + 2 * nlength\ (\text{clause-n clause}!\ t)$  **and**  $t < length\ (\text{clause-n clause})$

**shows**  $\text{transforms } tmL1\ (tpsL\ t)\ ttt\ (tpsL1\ t)$

*<proof>*

**definition**  $tpsL2 :: \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsL2\ t \equiv tps0$

$[j2 := nltape'\ (\text{clause-n clause})\ (\text{Suc } t),$

$j3 := (\lfloor \text{sat-take } t \rfloor_B, 1),$

$j3 + 1 := (\lfloor \text{literal-n } (\text{clause}!\ t) \rfloor_N, 1),$

$j3 + 2 := (\lfloor \text{literal-n } (\text{clause}!\ t)\ \text{mod } 2 \rfloor_N, 1)]$

**lemma**  $tmL2$  [*transforms-intros*]:

**assumes**  $ttt = 12 + 2 * nlength\ (\text{clause-n clause}!\ t) + 1$

**and**  $t < \text{length } (\text{clause-n clause})$   
**shows**  $\text{transforms } \text{tmL2 } (\text{tpsL } t) \text{ ttt } (\text{tpsL2 } t)$   
 ⟨proof⟩

**definition**  $\text{tpsL3} :: \text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL3 } t \equiv \text{tps0}$   
 $[j2 := \text{nltape}' (\text{clause-n clause}) (\text{Suc } t),$   
 $j3 := (\lfloor \text{sat-take } t \rfloor_B, 1),$   
 $j3 + 1 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ div } 2 \rfloor_N, 1),$   
 $j3 + 2 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ mod } 2 \rfloor_N, 1)]$

**lemma**  $\text{tmL3}$  [transforms-intros]:

**assumes**  $\text{ttt} = 16 + 4 * \text{nlength } (\text{clause-n clause ! } t)$   
**and**  $t < \text{length } (\text{clause-n clause})$   
**shows**  $\text{transforms } \text{tmL3 } (\text{tpsL } t) \text{ ttt } (\text{tpsL3 } t)$   
 ⟨proof⟩

**definition**  $\text{tpsL4} :: \text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL4 } t \equiv \text{tps0}$   
 $[j2 := \text{nltape}' (\text{clause-n clause}) (\text{Suc } t),$   
 $j3 := (\lfloor \text{sat-take } t \rfloor_B, 1),$   
 $j3 + 1 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ div } 2 \rfloor_N, 1),$   
 $j3 + 2 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ mod } 2 \rfloor_N, 1),$   
 $j3 + 3 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ div } 2 \in \text{set vars} \rfloor_B, 1)]$

**lemma**  $\text{tmL4}$  [transforms-intros]:

**assumes**  $\text{ttt} = 20 + 4 * \text{nlength } (\text{clause-n clause ! } t) + 67 * (\text{nlength vars})^2$   
**and**  $t < \text{length } (\text{clause-n clause})$   
**shows**  $\text{transforms } \text{tmL4 } (\text{tpsL } t) \text{ ttt } (\text{tpsL4 } t)$   
 ⟨proof⟩

**definition**  $\text{tpsL5} :: \text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL5 } t \equiv \text{tps0}$   
 $[j2 := \text{nltape}' (\text{clause-n clause}) (\text{Suc } t),$   
 $j3 := (\lfloor \text{sat-take } (\text{Suc } t) \rfloor_B, 1),$   
 $j3 + 1 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ div } 2 \rfloor_N, 1),$   
 $j3 + 2 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ mod } 2 \rfloor_N, 1),$   
 $j3 + 3 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ div } 2 \in \text{set vars} \rfloor_B, 1)]$

**lemma**  $\text{tmI}$  [transforms-intros]:

**assumes**  $\text{ttt} = 16$  **and**  $t < \text{length } (\text{clause-n clause})$   
**shows**  $\text{transforms } \text{tmI } (\text{tpsL4 } t) \text{ ttt } (\text{tpsL5 } t)$   
 ⟨proof⟩

**lemma**  $\text{tmL5}$  [transforms-intros]:

**assumes**  $\text{ttt} = 36 + 4 * \text{nlength } (\text{clause-n clause ! } t) + 67 * (\text{nlength vars})^2$   
**and**  $t < \text{length } (\text{clause-n clause})$   
**shows**  $\text{transforms } \text{tmL5 } (\text{tpsL } t) \text{ ttt } (\text{tpsL5 } t)$   
 ⟨proof⟩

**definition**  $\text{tpsL6} :: \text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL6 } t \equiv \text{tps0}$   
 $[j2 := \text{nltape}' (\text{clause-n clause}) (\text{Suc } t),$   
 $j3 := (\lfloor \text{sat-take } (\text{Suc } t) \rfloor_B, 1),$   
 $j3 + 1 := (\lfloor 0 \rfloor_N, 1),$   
 $j3 + 2 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ mod } 2 \rfloor_N, 1),$   
 $j3 + 3 := (\lfloor \text{literal-n } (\text{clause ! } t) \text{ div } 2 \in \text{set vars} \rfloor_B, 1)]$

**lemma**  $\text{tmL6}$  [transforms-intros]:

**assumes**  $\text{ttt} = 46 + 4 * \text{nlength } (\text{clause-n clause ! } t) + 67 * (\text{nlength vars})^2 + 2 * \text{nlength } (\text{literal-n } (\text{clause ! } t) \text{ div } 2)$   
**and**  $t < \text{length } (\text{clause-n clause})$   
**shows**  $\text{transforms } \text{tmL6 } (\text{tpsL } t) \text{ ttt } (\text{tpsL6 } t)$

*<proof>*

**definition** *tpsL7* :: *nat*  $\Rightarrow$  *tape list* **where**

*tpsL7 t*  $\equiv$  *tps0*  
[*j2* := *nltape'* (*clause-n clause*) (*Suc t*),  
*j3* := ( $\lfloor \text{sat-take } (\text{Suc } t) \rfloor_B, 1$ ),  
*j3* + 1 := ( $\lfloor 0 \rfloor_N, 1$ ),  
*j3* + 2 := ( $\lfloor 0 \rfloor_N, 1$ ),  
*j3* + 3 := ( $\lfloor \text{literal-n } (\text{clause} ! t) \text{ div } 2 \in \text{set vars} \rfloor_B, 1$ )]

**lemma** *tmL7* [*transforms-intros*]:

**assumes**  $ttt = 56 + 4 * nlength (\text{clause-n clause} ! t) + 67 * (nlength \text{vars})^2 + 2 * nlength (\text{literal-n } (\text{clause} ! t) \text{ div } 2) +$

$2 * nlength (\text{literal-n } (\text{clause} ! t) \text{ mod } 2)$

**and**  $t < length (\text{clause-n clause})$

**shows** *transforms tmL7 (tpsL t) ttt (tpsL7 t)*

*<proof>*

**definition** *tpsL8* :: *nat*  $\Rightarrow$  *tape list* **where**

*tpsL8 t*  $\equiv$  *tps0*  
[*j2* := *nltape'* (*clause-n clause*) (*Suc t*),  
*j3* := ( $\lfloor \text{sat-take } (\text{Suc } t) \rfloor_B, 1$ ),  
*j3* + 1 := ( $\lfloor 0 \rfloor_N, 1$ ),  
*j3* + 2 := ( $\lfloor 0 \rfloor_N, 1$ ),  
*j3* + 3 := ( $\lfloor 0 \rfloor_N, 1$ )]

**lemma** *tmL8*:

**assumes**  $ttt = 66 + 4 * nlength (\text{clause-n clause} ! t) + 67 * (nlength \text{vars})^2 +$

$2 * nlength (\text{literal-n } (\text{clause} ! t) \text{ div } 2) +$

$2 * nlength (\text{literal-n } (\text{clause} ! t) \text{ mod } 2) +$

$2 * nlength (\text{if literal-n } (\text{clause} ! t) \text{ div } 2 \in \text{set vars then } 1 \text{ else } 0)$

**and**  $t < length (\text{clause-n clause})$

**shows** *transforms tmL8 (tpsL t) ttt (tpsL8 t)*

*<proof>*

**lemma** *tmL8'*:

**assumes**  $ttt = 70 + 6 * nlength (\text{clause-n clause}) + 67 * (nlength \text{vars})^2$

**and**  $t < length (\text{clause-n clause})$

**shows** *transforms tmL8 (tpsL t) ttt (tpsL8 t)*

*<proof>*

**definition** *tpsL8'* :: *nat*  $\Rightarrow$  *tape list* **where**

*tpsL8' t*  $\equiv$  *tps0*  
[*j2* := *nltape'* (*clause-n clause*) (*Suc t*),  
*j3* := ( $\lfloor \text{sat-take } (\text{Suc } t) \rfloor_B, 1$ )]

**lemma** *tpsL8'*: *tpsL8' = tpsL8*

*<proof>*

**lemma** *tmL8''* [*transforms-intros*]:

**assumes**  $ttt = 70 + 6 * nlength (\text{clause-n clause}) + 67 * (nlength \text{vars})^2$

**and**  $t < length (\text{clause-n clause})$

**shows** *transforms tmL8 (tpsL t) ttt (tpsL8' t)*

*<proof>*

**lemma** *tmL* [*transforms-intros*]:

**assumes**  $ttt = length (\text{clause-n clause}) * (72 + 6 * nlength (\text{clause-n clause}) + 67 * (nlength \text{vars})^2) + 1$

**shows** *transforms tmL (tpsL 0) ttt (tpsL (length (\text{clause-n clause})))*

*<proof>*

**definition** *tps1* :: *tape list* **where**

*tps1*  $\equiv$  *tps0*  
[*j2* := *nltape'* (*clause-n clause*) (*length (\text{clause-n clause})*)],

$j_3 := (\lfloor \text{satisfies-clause } (\lambda v. v \in \text{set vars}) \text{ clause} \rfloor_B, 1)$

**lemma** *tps1*:  $\text{tps1} = \text{tpsL } (\text{length } (\text{clause-n clause}))$   
 ⟨proof⟩

**lemma** *tm1* [*transforms-intros*]:

**assumes**  $\text{ttt} = \text{length } (\text{clause-n clause}) * (72 + 6 * \text{nlength } (\text{clause-n clause}) + 67 * (\text{nlength vars})^2) + 1$   
**shows** *transforms tmL tps0 ttt tps1*  
 ⟨proof⟩

**definition** *tps2* :: *tape list where*

$\text{tps2} \equiv \text{tps0}$   
 $[j_2 := \text{nltape}' (\text{clause-n clause}) 0,$   
 $j_3 := (\lfloor \text{satisfies-clause } (\lambda v. v \in \text{set vars}) \text{ clause} \rfloor_B, 1)]$

**lemma** *tm2*:

**assumes**  $\text{ttt} = \text{length } (\text{clause-n clause}) * (72 + 6 * \text{nlength } (\text{clause-n clause}) + 67 * (\text{nlength vars})^2) + \text{nlength } (\text{clause-n clause}) + 4$   
**shows** *transforms tm2 tps0 ttt tps2*  
 ⟨proof⟩

**definition** *tps2'* :: *tape list where*

$\text{tps2}' \equiv \text{tps0}$   
 $[j_3 := (\lfloor \text{satisfies-clause } (\lambda v. v \in \text{set vars}) \text{ clause} \rfloor_B, 1)]$

**lemma** *tm2'*:

**assumes**  $\text{ttt} = 79 * (\text{nlength } (\text{clause-n clause}))^2 + 67 * (\text{nlength } (\text{clause-n clause})) * \text{nlength vars}^2 + 4$   
**shows** *transforms tm2 tps0 ttt tps2'*  
 ⟨proof⟩

**end**

**end**

**lemma** *transforms-tm-sat-clauseI* [*transforms-intros*]:

**fixes**  $j_1 j_2 j_3 :: \text{tapeidx}$   
**fixes**  $\text{tps tps}' :: \text{tape list}$  **and**  $\text{ttt } k :: \text{nat}$  **and**  $\text{vars} :: \text{nat list}$  **and**  $\text{clause} :: \text{literal list}$   
**assumes**  $0 < j_1 j_1 \neq j_2 j_3 + 5 < k j_1 < j_3 j_2 < j_3 0 < j_2 \text{length tps} = k$   
**assumes**  
 $\text{tps} ! j_1 = \text{nltape}' \text{ vars } 0$   
 $\text{tps} ! j_2 = \text{nltape}' (\text{clause-n clause}) 0$   
 $\text{tps} ! j_3 = (\lfloor 0 \rfloor_N, 1)$   
 $\text{tps} ! (j_3 + 1) = (\lfloor 0 \rfloor_N, 1)$   
 $\text{tps} ! (j_3 + 2) = (\lfloor 0 \rfloor_N, 1)$   
 $\text{tps} ! (j_3 + 3) = (\lfloor 0 \rfloor_N, 1)$   
 $\text{tps} ! (j_3 + 4) = (\lfloor 0 \rfloor_N, 1)$   
 $\text{tps} ! (j_3 + 5) = (\lfloor 0 \rfloor_N, 1)$

**assumes**  $\text{tps}' = \text{tps}$

$[j_3 := (\lfloor \text{satisfies-clause } (\lambda v. v \in \text{set vars}) \text{ clause} \rfloor_B, 1)]$

**assumes**  $\text{ttt} = 79 * (\text{nlength } (\text{clause-n clause}))^2 + 67 * (\text{nlength } (\text{clause-n clause})) * \text{nlength vars}^2 + 4$   
**shows** *transforms (tm-sat-clause j1 j2 j3) tps ttt tps'*  
 ⟨proof⟩

The following Turing machine expects a list of lists of numbers representing a formula  $\varphi$  on tape  $j_1$  and a list of numbers representing an assignment  $\alpha$  on tape  $j_2$ . It outputs on tape  $j_3$  the number 1 if  $\alpha$  satisfies  $\varphi$ , and otherwise the number 0. To do so the TM iterates over all clauses in  $\varphi$  and uses *tm-sat-clause* on each of them. It requires seven auxiliary tapes:  $j_3 + 1$  to store the clauses one at a time,  $j_3 + 2$  to store the results of *tm-sat-clause*, whose auxiliary tapes are  $j_3 + 3, \dots, j_3 + 7$ .

**definition** *tm-sat-formula* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine where*

*tm-sat-formula*  $j_1 j_2 j_3 \equiv$   
 $\text{tm-setn } j_3 1 ;;$

```

WHILE [] ; λrs. rs ! j1 ≠ □ DO
  tm-nextract # j1 (j3 + 1) ;;
  tm-sat-clause j2 (j3 + 1) (j3 + 2) ;;
  IF λrs. rs ! (j3 + 2) = □ THEN
    tm-setn j3 0
  ELSE
    []
  ENDIF ;;
  tm-erase-cr (j3 + 1) ;;
  tm-setn (j3 + 2) 0
DONE

```

**lemma** *tm-sat-formula-tm*:

**assumes**  $k \geq 2$  **and**  $G \geq 6$  **and**  $0 < j1$   $j1 \neq j2$   $j3 + 7 < k$   $j1 < j3$   $j2 < j3$   $0 < j2$   
**shows** *turing-machine*  $k$   $G$  (*tm-sat-formula*  $j1$   $j2$   $j3$ )  
*<proof>*

**locale** *turing-machine-sat-formula* =

**fixes**  $j1$   $j2$   $j3$  :: *tapeidx*

**begin**

**definition**  $tm1 \equiv tm\text{-setn } j3 \ 1$

**definition**  $tmL1 \equiv tm\text{-nextract } \# \ j1 \ (j3 + 1)$

**definition**  $tmL2 \equiv tmL1$  ;; *tm-sat-clause*  $j2$   $(j3 + 1)$   $(j3 + 2)$

**definition**  $tmI \equiv IF \ \lambda rs. \ rs \ ! \ (j3 + 2) = \square \ THEN \ tm\text{-setn } j3 \ 0 \ ELSE \ [] \ ENDIF$

**definition**  $tmL3 \equiv tmL2$  ;;  $tmI$

**definition**  $tmL4 \equiv tmL3$  ;; *tm-erase-cr*  $(j3 + 1)$

**definition**  $tmL5 \equiv tmL4$  ;; *tm-setn*  $(j3 + 2)$   $0$

**definition**  $tmL \equiv WHILE \ [] \ ; \ \lambda rs. \ rs \ ! \ j1 \ \neq \ \square \ DO \ tmL5 \ DONE$

**definition**  $tm2 \equiv tm1$  ;;  $tmL$

**lemma** *tm2-eq-tm-sat-formula*:  $tm2 = tm\text{-sat-formula } j1 \ j2 \ j3$

*<proof>*

**context**

**fixes**  $tps0$  :: *tape list* **and**  $k$  :: *nat* **and**  $vars$  :: *nat list* **and**  $\varphi$  :: *formula*

**assumes**  $jk$ :  $0 < j1$   $j1 \neq j2$   $j3 + 7 < k$   $j1 < j3$   $j2 < j3$   $0 < j2$  *length*  $tps0 = k$

**assumes**  $tps0$ :

$tps0 \ ! \ j1 = nlltape' \ (formula\text{-n } \varphi) \ 0$

$tps0 \ ! \ j2 = nltape' \ vars \ 0$

$tps0 \ ! \ j3 = ([0]_N, 1)$

$tps0 \ ! \ (j3 + 1) = ([[]]_{NL}, 1)$

$tps0 \ ! \ (j3 + 2) = ([0]_N, 1)$

$tps0 \ ! \ (j3 + 3) = ([0]_N, 1)$

$tps0 \ ! \ (j3 + 4) = ([0]_N, 1)$

$tps0 \ ! \ (j3 + 5) = ([0]_N, 1)$

$tps0 \ ! \ (j3 + 6) = ([0]_N, 1)$

$tps0 \ ! \ (j3 + 7) = ([0]_N, 1)$

**begin**

**definition**  $tps1 \equiv tps0$

$[j3 := ([1]_N, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:

**assumes**  $ttt = 12$

**shows** *transforms*  $tm1$   $tps0$   $ttt$   $tps1$

*<proof>*

**abbreviation** *sat-take*  $t \equiv (\lambda v. v \in \text{set } vars) \models \text{take } t \ \varphi$

**definition**  $tpsL$  :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL\ t \equiv tps0$   
 $[j1 := nlltape' (formula-n\ \varphi)\ t,$   
 $j3 := (\lfloor sat-take\ t \rfloor_B, 1)]$

**lemma**  $tpsL0$ :  $tpsL\ 0 = tps1$   
 $\langle proof \rangle$

**definition**  $tpsL1 :: nat \Rightarrow tape\ list$  **where**

$tpsL1\ t \equiv tps0$   
 $[j1 := nlltape' (formula-n\ \varphi)\ (Suc\ t),$   
 $j3 := (\lfloor sat-take\ t \rfloor_B, 1),$   
 $j3 + 1 := (\lfloor formula-n\ \varphi ! t \rfloor_{NL}, 1)]$

**lemma**  $tmL1$  [*transforms-intros*]:  
**assumes**  $ttt = 12 + 2 * nllength (formula-n\ \varphi ! t)$  **and**  $t < length (formula-n\ \varphi)$   
**shows** *transforms*  $tmL1 (tpsL\ t) ttt (tpsL1\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL2 :: nat \Rightarrow tape\ list$  **where**

$tpsL2\ t \equiv tps0$   
 $[j1 := nlltape' (formula-n\ \varphi)\ (Suc\ t),$   
 $j3 := (\lfloor sat-take\ t \rfloor_B, 1),$   
 $j3 + 1 := (\lfloor formula-n\ \varphi ! t \rfloor_{NL}, 1),$   
 $j3 + 2 := (\lfloor satisfies-clause\ (\lambda v. v \in set\ vars)\ (\varphi ! t) \rfloor_B, 1)]$

**lemma**  $tmL2$  [*transforms-intros*]:  
**assumes**  $ttt = 12 + 2 * nllength (formula-n\ \varphi ! t) +$   
 $(79 * (nllength (formula-n\ \varphi ! t))^2 +$   
 $67 * nllength (formula-n\ \varphi ! t) * (nllength\ vars)^2 + 4)$   
**and**  $t < length (formula-n\ \varphi)$   
**shows** *transforms*  $tmL2 (tpsL\ t) ttt (tpsL2\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL3 :: nat \Rightarrow tape\ list$  **where**

$tpsL3\ t \equiv tps0$   
 $[j1 := nlltape' (formula-n\ \varphi)\ (Suc\ t),$   
 $j3 := (\lfloor sat-take\ (Suc\ t) \rfloor_B, 1),$   
 $j3 + 1 := (\lfloor formula-n\ \varphi ! t \rfloor_{NL}, 1),$   
 $j3 + 2 := (\lfloor satisfies-clause\ (\lambda v. v \in set\ vars)\ (\varphi ! t) \rfloor_B, 1)]$

**lemma**  $tmI$  [*transforms-intros*]:  
**assumes**  $ttt = 16$  **and**  $t < length (formula-n\ \varphi)$   
**shows** *transforms*  $tmI (tpsL2\ t) ttt (tpsL3\ t)$   
 $\langle proof \rangle$

**lemma**  $tmL3$  [*transforms-intros*]:  
**assumes**  $ttt = 32 + 2 * nllength (formula-n\ \varphi ! t) +$   
 $79 * (nllength (formula-n\ \varphi ! t))^2 +$   
 $67 * nllength (formula-n\ \varphi ! t) * (nllength\ vars)^2$   
**and**  $t < length (formula-n\ \varphi)$   
**shows** *transforms*  $tmL3 (tpsL\ t) ttt (tpsL3\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL4 :: nat \Rightarrow tape\ list$  **where**

$tpsL4\ t \equiv tps0$   
 $[j1 := nlltape' (formula-n\ \varphi)\ (Suc\ t),$   
 $j3 := (\lfloor sat-take\ (Suc\ t) \rfloor_B, 1),$   
 $j3 + 1 := (\lfloor \rfloor_{NL}, 1),$   
 $j3 + 2 := (\lfloor satisfies-clause\ (\lambda v. v \in set\ vars)\ (\varphi ! t) \rfloor_B, 1)]$

**lemma**  $tmL4$  [*transforms-intros*]:  
**assumes**  $ttt = 39 + 4 * nllength (formula-n\ \varphi ! t) +$   
 $79 * (nllength (formula-n\ \varphi ! t))^2 +$

$67 * \text{nllength} (\text{formula-}n \varphi ! t) * (\text{nllength vars})^2$   
**and**  $t < \text{length} (\text{formula-}n \varphi)$   
**shows** *transforms tmL4 (tpsL t) ttt (tpsL4 t)*  
 ⟨proof⟩

**definition** *tpsL5 :: nat ⇒ tape list where*

$\text{tpsL5 } t \equiv \text{tps0}$   
 $[j1 := \text{nlltape}' (\text{formula-}n \varphi) (\text{Suc } t),$   
 $j3 := (\text{[sat-take (Suc } t)]_B, 1),$   
 $j3 + 1 := (\text{[[]]}_{NL}, 1),$   
 $j3 + 2 := (\text{[0]}_N, 1)]$

**lemma** *tmL5:*

**assumes**  $\text{ttt} = 49 + 4 * \text{nllength} (\text{formula-}n \varphi ! t) +$   
 $79 * (\text{nllength} (\text{formula-}n \varphi ! t))^2 +$   
 $67 * \text{nllength} (\text{formula-}n \varphi ! t) * (\text{nllength vars})^2 +$   
 $2 * \text{nlength} (\text{if satisfies-clause } (\lambda v. v \in \text{set vars}) (\varphi ! t) \text{ then } 1 \text{ else } 0)$   
**and**  $t < \text{length} (\text{formula-}n \varphi)$   
**shows** *transforms tmL5 (tpsL t) ttt (tpsL5 t)*  
 ⟨proof⟩

**definition** *tpsL5' :: nat ⇒ tape list where*

$\text{tpsL5}' t \equiv \text{tps0}$   
 $[j1 := \text{nlltape}' (\text{formula-}n \varphi) (\text{Suc } t),$   
 $j3 := (\text{[sat-take (Suc } t)]_B, 1)]$

**lemma** *tpsL5': tpsL5' = tpsL5*

⟨proof⟩

**lemma** *tmL5' [transforms-intros]:*

**assumes**  $\text{ttt} = 51 + 83 * (\text{nllength} (\text{formula-}n \varphi))^2 +$   
 $67 * \text{nllength} (\text{formula-}n \varphi) * (\text{nllength vars})^2$   
**and**  $t < \text{length} (\text{formula-}n \varphi)$   
**shows** *transforms tmL5 (tpsL t) ttt (tpsL5' t)*  
 ⟨proof⟩

**lemma** *tmL [transforms-intros]:*

**assumes**  $\text{ttt} = \text{length} (\text{formula-}n \varphi) * (53 + 83 * (\text{nllength} (\text{formula-}n \varphi))^2 + 67 * \text{nllength} (\text{formula-}n \varphi)$   
 $* (\text{nllength vars})^2) + 1$   
**shows** *transforms tmL (tpsL 0) ttt (tpsL (length (formula-n φ)))*  
 ⟨proof⟩

**lemma** *tm2:*

**assumes**  $\text{ttt} = \text{length} (\text{formula-}n \varphi) * (53 + 83 * (\text{nllength} (\text{formula-}n \varphi))^2 + 67 * \text{nllength} (\text{formula-}n \varphi)$   
 $* (\text{nllength vars})^2) + 13$   
**shows** *transforms tm2 tps0 ttt (tpsL (length (formula-n φ)))*  
 ⟨proof⟩

**definition** *tps2 :: tape list where*

$\text{tps2} \equiv \text{tps0}$   
 $[j1 := \text{nlltape} (\text{formula-}n \varphi),$   
 $j3 := (\text{[(λv. v ∈ set vars) = φ]}_B, 1)]$

**lemma** *tps2: tps2 = tpsL (length (formula-n φ))*

⟨proof⟩

**lemma** *tm2':*

**assumes**  $\text{ttt} = \text{length} (\text{formula-}n \varphi) * (53 + 83 * (\text{nllength} (\text{formula-}n \varphi))^2 + 67 * \text{nllength} (\text{formula-}n \varphi)$   
 $* (\text{nllength vars})^2) + 13$   
**shows** *transforms tm2 tps0 ttt tps2*  
 ⟨proof⟩

**end**

end

**lemma** *transforms-tm-sat-formulaI* [*transforms-intros*]:

**fixes** *j1 j2 j3* :: *tapeidx*

**fixes** *tps tps'* :: *tape list* **and** *ttt k* :: *nat* **and** *vars* :: *nat list* **and**  $\varphi$  :: *formula*

**assumes**  $0 < j1$   $j1 \neq j2$   $j3 + 7 < k$   $j1 < j3$   $j2 < j3$   $0 < j2$  *length tps = k*

**assumes**

*tps* ! *j1* = *nlltape'* (*formula-n*  $\varphi$ ) 0

*tps* ! *j2* = *nltape'* *vars* 0

*tps* ! *j3* = ( $\lfloor 0 \rfloor_N, 1$ )

*tps* ! (*j3* + 1) = ( $\lfloor \lfloor \rfloor_{NL}, 1$ )

*tps* ! (*j3* + 2) = ( $\lfloor 0 \rfloor_N, 1$ )

*tps* ! (*j3* + 3) = ( $\lfloor 0 \rfloor_N, 1$ )

*tps* ! (*j3* + 4) = ( $\lfloor 0 \rfloor_N, 1$ )

*tps* ! (*j3* + 5) = ( $\lfloor 0 \rfloor_N, 1$ )

*tps* ! (*j3* + 6) = ( $\lfloor 0 \rfloor_N, 1$ )

*tps* ! (*j3* + 7) = ( $\lfloor 0 \rfloor_N, 1$ )

**assumes** *tps'* = *tps*

[*j1* := *nlltape* (*formula-n*  $\varphi$ ),

*j3* := ( $\lfloor (\lambda v. v \in \text{set vars}) \models \varphi \rfloor_B, 1$ )]

**assumes** *ttt* = *length* (*formula-n*  $\varphi$ ) \* (53 + 83 \* (*nlllength* (*formula-n*  $\varphi$ ))<sup>2</sup> + 67 \* *nlllength* (*formula-n*  $\varphi$ ) \* (*nllength* *vars*)<sup>2</sup>) + 13

**shows** *transforms* (*tm-sat-formula* *j1 j2 j3*) *tps ttt tps'*

(*proof*)

### 4.2.3 A Turing machine for verifying SAT instances

The previous Turing machine, *tm-sat-formula*, expects a well-formed formula and a well-formed list representing an assignment on its tapes. The TM we ultimately need, however, is not guaranteed to be given anything well-formed as input and even the well-formed inputs require decoding from the binary alphabet to the quaternary alphabet used for lists of lists of numbers. The next TM takes care of all of that and, if everything was well-formed, runs *tm-sat-formula*. If the first element of the pair input is invalid, it outputs **1**, as required by the definition of SAT.

Thus, the next Turing machine implements the function *verify-sat* and therefore is a verifier for SAT.

**definition** *tm-verify-sat* :: *machine* **where**

*tm-verify-sat*  $\equiv$

*tm-right-many* {0..<sup>22</sup>} ;;

*tm-bindecode* 0 2 ;;

*tm-unpair* 2 3 4 ;;

*tm-even-length* 3 5 ;;

*tm-proper-symbols-lt* 3 6 4 ;;

*tm-and* 6 5 ;;

IF  $\lambda rs. rs ! 6 \neq \square$  THEN

*tm-bindecode* 3 7 ;;

*tm-numlistlist-wf* 7 8 ;;

IF  $\lambda rs. rs ! 8 \neq \square$  THEN

*tm-proper-symbols-lt* 4 10 4 ;;

IF  $\lambda rs. rs ! 10 \neq \square$  THEN

*tm-bindecode* 4 11 ;;

*tm-rstrip* # 11 ;;

*tm-numlist-wf* 11 12 ;;

IF  $\lambda rs. rs ! 12 \neq \square$  THEN

*tm-sat-formula* 7 11 14 ;;

*tm-copyn* 14 1

ELSE

$\square$

ENDIF

ELSE

$\square$

ENDIF

ELSE

```

    tm-setn 1 1
  ENDIF
ELSE
  tm-setn 1 1
ENDIF

```

**lemma** *tm-verify-sat-tm: turing-machine 22 6 tm-verify-sat*  
 ⟨*proof*⟩

**locale** *turing-machine-verify-sat*  
**begin**

**definition** *tm1*  $\equiv$  *tm-right-many* {0.. $<22$ }  
**definition** *tm2*  $\equiv$  *tm1* ;; *tm-bindecode* 0 2  
**definition** *tm3*  $\equiv$  *tm2* ;; *tm-unpair* 2 3 4  
**definition** *tm4*  $\equiv$  *tm3* ;; *tm-even-length* 3 5  
**definition** *tm5*  $\equiv$  *tm4* ;; *tm-proper-symbols-lt* 3 6 4  
**definition** *tm6*  $\equiv$  *tm5* ;; *tm-and* 6 5

**definition** *tmTTT1*  $\equiv$  *tm-bindecode* 4 11  
**definition** *tmTTT2*  $\equiv$  *tmTTT1* ;; *tm-rstrip* # 11  
**definition** *tmTTT3*  $\equiv$  *tmTTT2* ;; *tm-numlist-wf* 11 12

**definition** *tmTTTT1*  $\equiv$  *tm-sat-formula* 7 11 14  
**definition** *tmTTTT2*  $\equiv$  *tmTTTT1* ;; *tm-copyn* 14 1  
**definition** *tmTTTTI*  $\equiv$  *IF*  $\lambda$ rs. rs ! 12  $\neq$   $\square$  *THEN* *tmTTTT2* *ELSE* [] *ENDIF*

**definition** *tmTTT*  $\equiv$  *tmTTT3* ;; *tmTTTTI*  
**definition** *tmTTI*  $\equiv$  *IF*  $\lambda$ rs. rs ! 10  $\neq$   $\square$  *THEN* *tmTTT* *ELSE* [] *ENDIF*

**definition** *tmTT1*  $\equiv$  *tm-proper-symbols-lt* 4 10 4  
**definition** *tmTT*  $\equiv$  *tmTT1* ;; *tmTTI*  
**definition** *tmTI*  $\equiv$  *IF*  $\lambda$ rs. rs ! 8  $\neq$   $\square$  *THEN* *tmTT* *ELSE* *tm-setn* 1 1 *ENDIF*

**definition** *tmT1*  $\equiv$  *tm-bindecode* 3 7  
**definition** *tmT2*  $\equiv$  *tmT1* ;; *tm-numlistlist-wf* 7 8  
**definition** *tmT*  $\equiv$  *tmT2* ;; *tmTI*

**definition** *tmI*  $\equiv$  *IF*  $\lambda$ rs. rs ! 6  $\neq$   $\square$  *THEN* *tmT* *ELSE* *tm-setn* 1 1 *ENDIF*  
**definition** *tm7*  $\equiv$  *tm6* ;; *tmI*

**lemma** *tm7-eq-tm-verify-sat: tm7 = tm-verify-sat*  
 ⟨*proof*⟩

**context**  
**fixes** *tps0* :: *tape list* **and** *zs* :: *symbol list*  
**assumes** *zs: bit-symbols zs*  
**assumes** *tps0: tps0 = snd (start-config 22 zs)*  
**begin**

**definition** *tps1*  $\equiv$  *map* ( $\lambda$ tp. tp |#|= 1) *tps0*

**lemma** *map-upt-length: map f xs = map ( $\lambda$ i. f (xs ! i)) [0..*length xs*]*  
 ⟨*proof*⟩

**lemma** *tps1:*  
*tps1* ! 0 = ([*zs*], 1)  
 $0 < j \implies j < 22 \implies$  *tps1* ! *j* = ([[]], 1)  
*length tps1* = 22  
 ⟨*proof*⟩

**lemma** *tm1 [transforms-intros]: transforms tm1 tps0 1 tps1*  
 ⟨*proof*⟩

**definition**  $tps2 \equiv tps1$   
[2 := ( $\lfloor \text{bindecode } zs \rfloor$ , 1)]

**lemma**  $tm2$  [transforms-intros]:  
**assumes**  $ttt = 8 + 3 * \text{length } zs$   
**shows** transforms  $tm2$   $tps0$   $ttt$   $tps2$   
(proof)

**definition**  $tps3 \equiv tps1$   
[2 := ( $\lfloor \text{bindecode } zs \rfloor$ , 1),  
3 := ( $\lfloor \text{first } (\text{bindecode } zs) \rfloor$ , 1),  
4 := ( $\lfloor \text{second } (\text{bindecode } zs) \rfloor$ , 1)]

**lemma**  $tm3$  [transforms-intros]:  
**assumes**  $ttt = 21 + 3 * \text{length } zs + 6 * \text{length } (\text{bindecode } zs)$   
**shows** transforms  $tm3$   $tps0$   $ttt$   $tps3$   
(proof)

**definition**  $tps4 \equiv tps1$   
[2 := ( $\lfloor \text{bindecode } zs \rfloor$ , 1),  
3 := ( $\lfloor \text{first } (\text{bindecode } zs) \rfloor$ , 1),  
4 := ( $\lfloor \text{second } (\text{bindecode } zs) \rfloor$ , 1),  
5 := ( $\lfloor \text{even } (\text{length } (\text{first } (\text{bindecode } zs))) \rfloor_B$ , 1)]

**lemma**  $tm4$  [transforms-intros]:  
**assumes**  $ttt = 28 + 3 * \text{length } zs + 6 * \text{length } (\text{bindecode } zs) + 7 * \text{length } (\text{first } (\text{bindecode } zs))$   
**shows** transforms  $tm4$   $tps0$   $ttt$   $tps4$   
(proof)

**definition**  $tps5 \equiv tps1$   
[2 := ( $\lfloor \text{bindecode } zs \rfloor$ , 1),  
3 := ( $\lfloor \text{first } (\text{bindecode } zs) \rfloor$ , 1),  
4 := ( $\lfloor \text{second } (\text{bindecode } zs) \rfloor$ , 1),  
5 := ( $\lfloor \text{even } (\text{length } (\text{first } (\text{bindecode } zs))) \rfloor_B$ , 1),  
6 := ( $\lfloor \text{proper-symbols-}lt\ 4\ (\text{first } (\text{bindecode } zs)) \rfloor_B$ , 1)]

**lemma**  $tm5$  [transforms-intros]:  
**assumes**  $ttt = 33 + 3 * \text{length } zs + 6 * \text{length } (\text{bindecode } zs) + 14 * \text{length } (\text{first } (\text{bindecode } zs))$   
**shows** transforms  $tm5$   $tps0$   $ttt$   $tps5$   
(proof)

**abbreviation**  $ys \equiv \text{bindecode } zs$   
**abbreviation**  $xs \equiv \text{bindecode } (\text{first } ys)$   
**abbreviation**  $vs \equiv \text{rstrip } 5\ (\text{bindecode } (\text{second } ys))$

**definition**  $tps6 \equiv tps1$   
[2 := ( $\lfloor ys \rfloor$ , 1),  
3 := ( $\lfloor \text{first } ys \rfloor$ , 1),  
4 := ( $\lfloor \text{second } ys \rfloor$ , 1),  
5 := ( $\lfloor \text{even } (\text{length } (\text{first } ys)) \rfloor_B$ , 1),  
6 := ( $\lfloor \text{proper-symbols-}lt\ 4\ (\text{first } ys) \wedge \text{even } (\text{length } (\text{first } ys)) \rfloor_B$ , 1)]

**lemma**  $tm6$  [transforms-intros]:  
**assumes**  $ttt = 36 + 3 * \text{length } zs + 6 * \text{length } (\text{bindecode } zs) + 14 * \text{length } (\text{first } (\text{bindecode } zs))$   
**shows** transforms  $tm6$   $tps0$   $ttt$   $tps6$   
(proof)

**context**  
**assumes**  $bs\text{-}even$ :  $\text{proper-symbols-}lt\ 4\ (\text{first } ys) \wedge \text{even } (\text{length } (\text{first } ys))$   
**begin**

**lemma**  $bs$ :  $\text{bit-symbols } (\text{first } ys)$

*<proof>*

**definition**  $tpsT1 \equiv tps1$

$[2 := (\lfloor ys \rfloor, 1),$   
 $3 := (\lfloor first\ ys \rfloor, 1),$   
 $4 := (\lfloor second\ ys \rfloor, 1),$   
 $5 := (\lfloor even\ (length\ (first\ ys)) \rfloor_B, 1),$   
 $6 := (\lfloor proper\ symbols\ lt\ 4\ (first\ ys) \wedge even\ (length\ (first\ ys)) \rfloor_B, 1),$   
 $7 := (\lfloor bindecode\ (first\ ys) \rfloor, 1)]$

**lemma**  $tmT1$  [*transforms-intros*]:

**assumes**  $ttt = 7 + 3 * length\ (first\ ys)$   
**shows** *transforms*  $tmT1\ tps6\ ttt\ tpsT1$   
*<proof>*

**definition**  $tpsT2 \equiv tps1$

$[2 := (\lfloor ys \rfloor, 1),$   
 $3 := (\lfloor first\ ys \rfloor, 1),$   
 $4 := (\lfloor second\ ys \rfloor, 1),$   
 $5 := (\lfloor even\ (length\ (first\ ys)) \rfloor_B, 1),$   
 $6 := (\lfloor proper\ symbols\ lt\ 4\ (first\ ys) \wedge even\ (length\ (first\ ys)) \rfloor_B, 1),$   
 $7 := (\lfloor bindecode\ (first\ ys) \rfloor, 1),$   
 $8 := (\lfloor numlistlist\ wf\ (bindecode\ (first\ ys)) \rfloor_B, 1)]$

**lemma**  $tmT2$  [*transforms-intros*]:

**assumes**  $ttt = 213 + 3 * length\ (first\ ys) + 39 * length\ (bindecode\ (first\ ys))$   
**shows** *transforms*  $tmT2\ tps6\ ttt\ tpsT2$   
*<proof>*

**context**

**assumes** *first-wf*:  $numlistlist\ wf\ (bindecode\ (first\ ys))$

**begin**

**definition**  $tpsTT1 \equiv tps1$

$[2 := (\lfloor ys \rfloor, 1),$   
 $3 := (\lfloor first\ ys \rfloor, 1),$   
 $4 := (\lfloor second\ ys \rfloor, 1),$   
 $5 := (\lfloor even\ (length\ (first\ ys)) \rfloor_B, 1),$   
 $6 := (\lfloor proper\ symbols\ lt\ 4\ (first\ ys) \wedge even\ (length\ (first\ ys)) \rfloor_B, 1),$   
 $7 := (\lfloor bindecode\ (first\ ys) \rfloor, 1),$   
 $8 := (\lfloor numlistlist\ wf\ (bindecode\ (first\ ys)) \rfloor_B, 1),$   
 $10 := (\lfloor proper\ symbols\ lt\ 4\ (second\ ys) \rfloor_B, 1)]$

**lemma**  $tmTT1$  [*transforms-intros*]:

**assumes**  $ttt = 5 + 7 * length\ (second\ ys)$   
**shows** *transforms*  $tmTT1\ tpsT2\ ttt\ tpsTT1$   
*<proof>*

**context**

**assumes** *proper-second*:  $proper\ symbols\ lt\ 4\ (second\ ys)$

**begin**

**definition**  $tpsTTT1 \equiv tps1$

$[2 := (\lfloor ys \rfloor, 1),$   
 $3 := (\lfloor first\ ys \rfloor, 1),$   
 $4 := (\lfloor second\ ys \rfloor, 1),$   
 $5 := (\lfloor even\ (length\ (first\ ys)) \rfloor_B, 1),$   
 $6 := (\lfloor proper\ symbols\ lt\ 4\ (first\ ys) \wedge even\ (length\ (first\ ys)) \rfloor_B, 1),$   
 $7 := (\lfloor xs \rfloor, 1),$   
 $8 := (\lfloor numlistlist\ wf\ xs \rfloor_B, 1),$   
 $10 := (\lfloor proper\ symbols\ lt\ 4\ (second\ ys) \rfloor_B, 1),$   
 $11 := (\lfloor bindecode\ (second\ ys) \rfloor, 1)]$

**lemma** *tmTTT1* [*transforms-intros*]:  
**assumes**  $ttt = 7 + 3 * \text{length} (\text{second } ys)$   
**shows** *transforms tmTTT1 tpsTT1 ttt tpsTTT1*  
*<proof>*

**definition** *tpsTTT2*  $\equiv$  *tps1*

[2 := ( $\lfloor ys \rfloor$ , 1),  
3 := ( $\lfloor \text{first } ys \rfloor$ , 1),  
4 := ( $\lfloor \text{second } ys \rfloor$ , 1),  
5 := ( $\lfloor \text{even} (\text{length} (\text{first } ys)) \rfloor_B$ , 1),  
6 := ( $\lfloor \text{proper-symbols-lt } 4 (\text{first } ys) \wedge \text{even} (\text{length} (\text{first } ys)) \rfloor_B$ , 1),  
7 := ( $\lfloor xs \rfloor$ , 1),  
8 := ( $\lfloor \text{numlistlist-wf } xs \rfloor_B$ , 1),  
10 := ( $\lfloor \text{proper-symbols-lt } 4 (\text{second } ys) \rfloor_B$ , 1),  
11 := ( $\lfloor vs \rfloor$ , 1)]

**lemma** *tmTTT2* [*transforms-intros*]:  
**assumes**  $ttt = 12 + 3 * \text{length} (\text{second } ys) + 3 * \text{length} (\text{bindecode} (\text{second } ys))$   
**shows** *transforms tmTTT2 tpsTT1 ttt tpsTTT2*  
*<proof>*

**definition** *tpsTTT3*  $\equiv$  *tps1*

[2 := ( $\lfloor ys \rfloor$ , 1),  
3 := ( $\lfloor \text{first } ys \rfloor$ , 1),  
4 := ( $\lfloor \text{second } ys \rfloor$ , 1),  
5 := ( $\lfloor \text{even} (\text{length} (\text{first } ys)) \rfloor_B$ , 1),  
6 := ( $\lfloor \text{proper-symbols-lt } 4 (\text{first } ys) \wedge \text{even} (\text{length} (\text{first } ys)) \rfloor_B$ , 1),  
7 := ( $\lfloor xs \rfloor$ , 1),  
8 := ( $\lfloor \text{numlistlist-wf } xs \rfloor_B$ , 1),  
10 := ( $\lfloor \text{proper-symbols-lt } 4 (\text{second } ys) \rfloor_B$ , 1),  
11 := ( $\lfloor vs \rfloor$ , 1),  
12 := ( $\lfloor \text{numlist-wf } vs \rfloor_B$ , 1)]

**lemma** *tmTTT3* [*transforms-intros*]:  
**assumes**  $ttt = 106 + 3 * \text{length} (\text{second } ys) + 3 * \text{length} (\text{bindecode} (\text{second } ys)) + 19 * \text{length } vs$   
**shows** *transforms tmTTT3 tpsTT1 ttt tpsTTT3*  
*<proof>*

**context**

**assumes** *second-wf: numlist-wf vs*

**begin**

**definition** *tpsTTTT1*  $\equiv$  *tps1*

[2 := ( $\lfloor ys \rfloor$ , 1),  
3 := ( $\lfloor \text{first } ys \rfloor$ , 1),  
4 := ( $\lfloor \text{second } ys \rfloor$ , 1),  
5 := ( $\lfloor \text{even} (\text{length} (\text{first } ys)) \rfloor_B$ , 1),  
6 := ( $\lfloor \text{proper-symbols-lt } 4 (\text{first } ys) \wedge \text{even} (\text{length} (\text{first } ys)) \rfloor_B$ , 1),  
7 := ( $\lfloor xs \rfloor$ , 1),  
8 := ( $\lfloor \text{numlistlist-wf } xs \rfloor_B$ , 1),  
10 := ( $\lfloor \text{proper-symbols-lt } 4 (\text{second } ys) \rfloor_B$ , 1),  
11 := ( $\lfloor vs \rfloor$ , 1),  
12 := ( $\lfloor \text{numlist-wf } vs \rfloor_B$ , 1),  
7 := *nlltape (formula-n (zs-formula xs))*,  
14 := ( $\lfloor (\lambda v. v \in \text{set} (\text{zs-numlist } vs)) \models \text{zs-formula } xs \rfloor_B$ , 1)]

**lemma** *tmTTTT1* [*transforms-intros*]:  
**assumes**  $ttt = \text{length} (\text{formula-n } (\text{zs-formula } xs)) * (53 + 83 * (\text{nllength} (\text{formula-n } (\text{zs-formula } xs)))^2 + 67 * \text{nllength} (\text{formula-n } (\text{zs-formula } xs)) * (\text{nllength} (\text{zs-numlist } vs))^2) + 13$   
**shows** *transforms tmTTTT1 tpsTTT3 ttt tpsTTTT1*  
*<proof>*

**definition**  $tpsTTTT2 \equiv tps1$

$[1 := (\lfloor (\lambda v. v \in \text{set } (zs\text{-numlist } vs)) \models zs\text{-formula } xs \rfloor_B, 1),$   
 $2 := (\lfloor ys \rfloor, 1),$   
 $3 := (\lfloor \text{first } ys \rfloor, 1),$   
 $4 := (\lfloor \text{second } ys \rfloor, 1),$   
 $5 := (\lfloor \text{even } (\text{length } (\text{first } ys)) \rfloor_B, 1),$   
 $6 := (\lfloor \text{proper-symbols-lt } 4 (\text{first } ys) \wedge \text{even } (\text{length } (\text{first } ys)) \rfloor_B, 1),$   
 $7 := (\lfloor xs \rfloor, 1),$   
 $8 := (\lfloor \text{numlistlist-wf } xs \rfloor_B, 1),$   
 $10 := (\lfloor \text{proper-symbols-lt } 4 (\text{second } ys) \rfloor_B, 1),$   
 $11 := (\lfloor vs \rfloor, 1),$   
 $12 := (\lfloor \text{numlist-wf } vs \rfloor_B, 1),$   
 $7 := \text{nlltape } (\text{formula-n } (zs\text{-formula } xs)),$   
 $14 := (\lfloor (\lambda v. v \in \text{set } (zs\text{-numlist } vs)) \models zs\text{-formula } xs \rfloor_B, 1)]$

**lemma**  $tmTTTT2$ :

**assumes**  $ttt = \text{length } (\text{formula-n } (zs\text{-formula } xs)) *$   
 $(53 + 83 * (\text{nllength } (\text{formula-n } (zs\text{-formula } xs)))^2 + 67 * \text{nllength } (\text{formula-n } (zs\text{-formula } xs)) * (\text{nllength}$   
 $(zs\text{-numlist } vs))^2) +$   
 $27 + 3 * (\text{nlength } (\text{if } (\lambda v. v \in \text{set } (zs\text{-numlist } vs)) \models zs\text{-formula } xs \text{ then } 1 \text{ else } 0))$   
**shows**  $\text{transforms } tmTTTT2 \text{ } tpsTTTT3 \text{ } ttt \text{ } tpsTTTT2$   
 $\langle \text{proof} \rangle$

**lemma**  $tmTTTT2'$  [*transforms-intros*]:

**assumes**  $ttt = 203 * \text{length } zs \wedge 4 + 30$   
**shows**  $\text{transforms } tmTTTT2 \text{ } tpsTTTT3 \text{ } ttt \text{ } tpsTTTT2$   
 $\langle \text{proof} \rangle$

**end**

**definition**  $tpsTTT \equiv (\text{if } \text{numlist-wf } vs \text{ then } tpsTTTT2 \text{ else } tpsTTTT3)$

**lemma**  $\text{length-tpsTTT}$ :  $\text{length } tpsTTT = 22$   
 $\langle \text{proof} \rangle$

**lemma**  $tpsTTT$ :  $tpsTTT ! 1 =$

$(\lfloor \text{if } \text{numlist-wf } vs \text{ then } (\text{if } (\lambda v. v \in \text{set } (zs\text{-numlist } vs)) \models zs\text{-formula } xs \text{ then } 1 \text{ else } 0) \text{ else } 0 \rfloor_N, 1)$   
 $\langle \text{proof} \rangle$

**lemma**  $tmTTTI$  [*transforms-intros*]:

**assumes**  $ttt = 203 * \text{length } zs \wedge 4 + 32$   
**shows**  $\text{transforms } tmTTTI \text{ } tpsTTTT3 \text{ } ttt \text{ } tpsTTT$   
 $\langle \text{proof} \rangle$

**lemma**  $tmTTT$ :

**assumes**  $ttt = 138 + 3 * \text{length } (\text{second } ys) + 3 * \text{length } (\text{bindecode } (\text{second } ys)) +$   
 $19 * \text{length } vs + 203 * \text{length } zs \wedge 4$   
**shows**  $\text{transforms } tmTTT \text{ } tpsTT1 \text{ } ttt \text{ } tpsTTT$   
 $\langle \text{proof} \rangle$

**lemma**  $tmTTT'$  [*transforms-intros*]:

**assumes**  $ttt = 138 + 228 * \text{length } zs \wedge 4$   
**shows**  $\text{transforms } tmTTT \text{ } tpsTT1 \text{ } ttt \text{ } tpsTTT$   
 $\langle \text{proof} \rangle$

**end**

**definition**  $tpsTT \equiv (\text{if } \text{proper-symbols-lt } 4 (\text{second } ys) \text{ then } tpsTTT \text{ else } tpsTT1)$

**lemma**  $\text{length-tpsTT}$ :  $\text{length } tpsTT = 22$   
 $\langle \text{proof} \rangle$

**lemma** *tpsTT*:  $tpsTT ! 1 =$   
 (ncontents  
 (if proper-symbols-lt 4 (second ys)  $\wedge$  numlist-wf vs  
 then if ( $\lambda v. v \in \text{set } (zs\text{-numlist } vs)$ )  $\models$  zs-formula xs then 1 else 0  
 else 0),  
 1)  
 <proof>

**lemma** *tmTTI* [transforms-intros]:  
**assumes**  $ttt = 140 + 228 * \text{length } zs \wedge 4$   
**shows** transforms *tmTTI* *tpsTT1* *t* *tpsTT*  
 <proof>

**lemma** *tmTT* [transforms-intros]:  
**assumes**  $ttt = 145 + 7 * \text{length } (\text{second } ys) + 228 * \text{length } zs \wedge 4$   
**shows** transforms *tmTT* *tpsT2* *t* *tpsTT*  
 <proof>

end

**definition** *tpsTE*  $\equiv$  *tps1*  
 [2 := ( $\lfloor ys \rfloor$ , 1),  
 3 := ( $\lfloor \text{first } ys \rfloor$ , 1),  
 4 := ( $\lfloor \text{second } ys \rfloor$ , 1),  
 5 := ( $\lfloor \text{even } (\text{length } (\text{first } ys)) \rfloor_B$ , 1),  
 6 := ( $\lfloor \text{proper-symbols-lt } 4 (\text{first } ys) \wedge \text{even } (\text{length } (\text{first } ys)) \rfloor_B$ , 1),  
 7 := ( $\lfloor \text{bindecode } (\text{first } ys) \rfloor$ , 1),  
 8 := ( $\lfloor \text{numlistlist-wf } xs \rfloor_B$ , 1),  
 1 := ( $\lfloor 1 \rfloor_N$ , 1)]

**definition** *tpsT*  $\equiv$  (if numlistlist-wf xs then *tpsTT* else *tpsTE*)

**lemma** *length-tpsT*:  $\text{length } tpsT = 22$   
 <proof>

**lemma** *tpsT*:  $tpsT ! 1 =$   
 (ncontents  
 (if numlistlist-wf xs  
 then if proper-symbols-lt 4 (second ys)  $\wedge$  numlist-wf vs  
 then if ( $\lambda v. v \in \text{set } (zs\text{-numlist } vs)$ )  $\models$  zs-formula xs then 1 else 0  
 else 0),  
 1)  
 <proof>

**lemma** *tmTI* [transforms-intros]:  
**assumes**  $ttt = 147 + 7 * \text{length } (\text{second } ys) + 228 * \text{length } zs \wedge 4$   
**shows** transforms *tmTI* *tpsT2* *t* *tpsT*  
 <proof>

**lemma** *tmT* [transforms-intros]:  
**assumes**  $ttt = 360 + 3 * \text{length } (\text{first } ys) + 39 * \text{length } xs + 7 * \text{length } (\text{second } ys) + 228 * \text{length } zs \wedge 4$   
**shows** transforms *tmT* *tps6* *t* *tpsT*  
 <proof>

end

**definition** *tpsE*  $\equiv$  *tps1*  
 [2 := ( $\lfloor ys \rfloor$ , 1),  
 3 := ( $\lfloor \text{first } ys \rfloor$ , 1),  
 4 := ( $\lfloor \text{second } ys \rfloor$ , 1),  
 5 := ( $\lfloor \text{even } (\text{length } (\text{first } ys)) \rfloor_B$ , 1),  
 6 := ( $\lfloor \text{proper-symbols-lt } 4 (\text{first } ys) \wedge \text{even } (\text{length } (\text{first } ys)) \rfloor_B$ , 1),

$1 := ([1]_N, 1)$

**definition**  $tps7 \equiv$  (if proper-symbols- $lt\ 4$  (first  $ys$ )  $\wedge$  even (length (first  $ys$ )) then  $tpsT$  else  $tpsE$ )

**lemma**  $length\ tps7$ :  $length\ tps7 = 22$   
 ⟨proof⟩

**lemma**  $tps7$ :  $tps7 ! 1 =$   
 (ncontents  
 (if proper-symbols- $lt\ 4$  (first  $ys$ )  $\wedge$  even (length (first  $ys$ ))  $\wedge$  numlistlist-wf  $xs$   
 then if proper-symbols- $lt\ 4$  (second  $ys$ )  $\wedge$  numlist-wf  $vs$   
 then if  $(\lambda v. v \in set\ (zs\ numlist\ vs)) \models zs\ formula\ xs$  then 1 else 0  
 else 0  
 else 1),  
 1)  
 ⟨proof⟩

**lemma**  $tps7'$ :  $tps7 ! 1 = ([verify\ sat\ zs], 1)$   
 ⟨proof⟩

**lemma**  $tmI$  [transforms-intros]:  
**assumes**  $ttt = 362 + 3 * length\ (first\ ys) + 39 * length\ xs + 7 * length\ (second\ ys) + 228 * length\ zs \wedge 4$   
**shows** transforms  $tmI\ tps6\ ttt\ tps7$   
 ⟨proof⟩

**lemma**  $tm7$ :  
**assumes**  $ttt = 398 + 3 * length\ zs + 6 * length\ ys + 17 * length\ (first\ ys) +$   
 $39 * length\ xs + 7 * length\ (second\ ys) + 228 * length\ zs \wedge 4$   
**shows** transforms  $tm7\ tps0\ ttt\ tps7$   
 ⟨proof⟩

**lemma**  $tm7'$  [transforms-intros]:  
**assumes**  $ttt = 398 + 300 * length\ zs \wedge 4$   
**shows** transforms  $tm7\ tps0\ ttt\ tps7$   
 ⟨proof⟩

end

end

**lemma** transforms- $tm\ verify\ sat$ :  
**fixes**  $zs :: symbol\ list$  **and**  $tps :: tape\ list$   
**assumes** bit-symbols  $zs$   
**and**  $tps = snd\ (start\ config\ 22\ zs)$   
**and**  $ttt = 398 + 300 * length\ zs \wedge 4$   
**shows**  $\exists\ tps'.\ tps' ! 1 = ([verify\ sat\ zs], 1) \wedge$  transforms  $tm\ verify\ sat\ tps\ ttt\ tps'$   
 ⟨proof⟩

With the Turing machine just constructed and the polynomial  $p(n) = n$  we can satisfy the definition of  $\mathcal{NP}$  and prove the main result of this chapter.

**theorem**  $SAT\ in\ NP$ :  $SAT \in \mathcal{NP}$   
 ⟨proof⟩

end

# Chapter 5

## Obliviousness

In order to show that **SAT** is  $\mathcal{NP}$ -hard we will eventually show how to reduce an arbitrary language  $L \in \mathcal{NP}$  to **SAT**. The proof can only use properties of  $L$  common to all languages in  $\mathcal{NP}$ . The definition of  $\mathcal{NP}$  provides us with a verifier Turing machine  $M$  for  $L$ , of which we only know that it is running in polynomial time. In addition by lemma *NP-output-len-1* we can assume that  $M$  outputs a single bit symbol. In this chapter we are going to show that we can make additional assumptions about  $M$ , namely:

1.  $M$  has only two tapes.
2.  $M$  halts on  $\langle x, u \rangle$  with the output tape head on the symbol **1** iff.  $u$  is a certificate for  $x$ .
3.  $M$  is *oblivious*, which means that on any input  $x$  the head positions of  $M$  on all its tapes depend only on the *length* of  $x$ , not on the symbols in  $x$  [2, Remark 1.7].

These additional properties will somewhat simplify the reduction of  $L$  to **SAT**, more precisely the construction of the CNF formulas (see Chapter 6).

In order to achieve this goal we will show how to simulate any polynomial-time multi-tape TM in polynomial time on a two-tape oblivious TM that halts with the output tape head on cell 1.

Given a polynomial-time  $k$ -tape TM  $M$ , the basic approach is to construct a two-tape TM that encodes the  $k$  tapes of  $M$  on its output tape in such a way that every cell encodes  $k$  symbols of  $M$  and flags for  $M$ 's tape heads. This is the same idea as used by Dalvit and Thiemann [5] and originally Hartmanis and Stearns [9] for simulating a multi-tape TM on a single-tape TM. After all our two-tape simulator can only properly use a single tape (the output/work tape). This simulator has roughly a quadratic running time overhead and so keeps the running time polynomial. However, it is not generally an oblivious TM.

To make the simulator TM oblivious, we have it initially “format” a section on the output tape that is long enough to hold everything  $M$  is going to write and whose length only depends on the input length. To simulate one step of  $M$ , the simulator then sweeps its output tape head all the way from the start of the tape to the end of the formatted space and back again, moving one cell per step. During these sweeps it executes one step of the simulation of  $M$ . Since the size of the formatted space only depends on the input length, the simulator performs the same head movements on inputs of the same length, resulting in an oblivious behavior. Moreover, it is easy to make it halt with the output tape head on cell number 1.

The formatter TM is described in Section 5.2. The simulator TM is then constructed in Section 5.3. Finally Section 5.4 states the main result of this chapter.

Before any of this, however, we have to define some basic concepts surrounding obliviousness.

### 5.1 Oblivious Turing machines

```
theory Oblivious
  imports Memorizing
begin
```

This section provides us with the tools for showing that a Turing machine is oblivious and for combining oblivious TMs into more complex oblivious TMs.

So far our analysis of Turing machines involved their semantics and running time bounds. For this we mainly used the *transforms* predicate, which relates a start configuration and a halting configuration and

an upper bound for the running time of a TM to transit from the one configuration to the other. To deal with obliviousness, we need to look more closely and inspect the sequence of tape head positions during the TM's execution, rather than only the running time.

The subsections in this section roughly correspond to Sections 2.1 to 2.5. In the first subsection we introduce a predicate *trace* analogous to *transforms* and show its behavior under sequential composition of TMs and loops (we will not need branches). The next subsection shows the head position sequences for those few elementary TMs from Section 2.4 that we need for our more complex oblivious TMs later. These constructions will also heavily use the memorization-in-states technique from Section 2.5, which we adapt to this chapter's needs in the final subsection.

### 5.1.1 Traces and head positions

In order to show that a Turing machine is oblivious we need to keep track of its head positions. Consider a machine  $M$  that transits from a configuration  $cfg1$  to a configuration  $cfg2$  in  $t$  steps. We call the sequence of head positions on the first two tapes a *trace*. If we ignore the initial head positions, the length of a trace equals  $t$ . Moreover we will only consider traces where  $M$  either does not halt or halts in the very last step. These two properties mean, for example, that we can simply concatenate a trace of a TM that halts and trace of another TM and get the trace of the sequential execution of both TMs. Similarly, analysing while loops is simplified by these two extra assumptions. The next predicate defines what it means for a list  $es$  to be a trace.

**definition** *trace* :: machine  $\Rightarrow$  config  $\Rightarrow$  (nat  $\times$  nat) list  $\Rightarrow$  config  $\Rightarrow$  bool **where**

*trace*  $M$   $cfg1$   $es$   $cfg2$   $\equiv$   
 $execute\ M\ cfg1\ (length\ es) = cfg2 \wedge$   
 $(\forall i < length\ es.\ fst\ (execute\ M\ cfg1\ i) < length\ M) \wedge$   
 $(\forall i < length\ es.\ execute\ M\ cfg1\ (Suc\ i) <\#\> 0 = fst\ (es\ !\ i)) \wedge$   
 $(\forall i < length\ es.\ execute\ M\ cfg1\ (Suc\ i) <\#\> 1 = snd\ (es\ !\ i))$

We will consider traces for machines with more than two tapes, too, but only for auxiliary constructions in combination with the memorizing-in-states technique. Therefore our definition is limited to start configurations with two tapes. A machine is *oblivious* if there is a function mapping the input length to the trace that takes the machine from the start configuration with that input to a halting configuration.

**definition** *oblivious* :: machine  $\Rightarrow$  bool **where**

*oblivious*  $M \equiv \exists e.$   
 $(\forall zs.\ bit\text{-symbols}\ zs \longrightarrow (\exists tps.\ trace\ M\ (start\text{-config}\ 2\ zs)\ (e\ (length\ zs))\ (length\ M,\ tps)))$

**lemma** *trace-Nil*: *trace*  $M$   $cfg$  []  $cfg$

*<proof>*

**lemma** *traceI*:

**assumes**  $execute\ M\ (q1,\ tps1)\ (length\ es) = (q2,\ tps2)$   
**and**  $\bigwedge i.\ i < length\ es \implies fst\ (execute\ M\ (q1,\ tps1)\ i) < length\ M$   
**and**  $\bigwedge i.\ i < length\ es \implies$   
 $execute\ M\ (q1,\ tps1)\ (Suc\ i) <\#\> 0 = fst\ (es\ !\ i) \wedge$   
 $execute\ M\ (q1,\ tps1)\ (Suc\ i) <\#\> 1 = snd\ (es\ !\ i)$   
**shows**  $trace\ M\ (q1,\ tps1)\ es\ (q2,\ tps2)$   
*<proof>*

**lemma** *traceI'*:

**assumes**  $execute\ M\ cfg1\ (length\ es) = cfg2$   
**and**  $\bigwedge i.\ i < length\ es \implies fst\ (execute\ M\ cfg1\ i) < length\ M$   
**and**  $\bigwedge i.\ i < length\ es \implies$   
 $execute\ M\ cfg1\ (Suc\ i) <\#\> 0 = fst\ (es\ !\ i) \wedge$   
 $execute\ M\ cfg1\ (Suc\ i) <\#\> 1 = snd\ (es\ !\ i)$   
**shows**  $trace\ M\ cfg1\ es\ cfg2$   
*<proof>*

**lemma** *trace-additive*:

**assumes**  $trace\ M\ (q1,\ tps1)\ es1\ (q2,\ tps2)$  **and**  $trace\ M\ (q2,\ tps2)\ es2\ (q3,\ tps3)$   
**shows**  $trace\ M\ (q1,\ tps1)\ (es1\ @\ es2)\ (q3,\ tps3)$   
*<proof>*

**lemma** *trace-additive'*:

**assumes** *trace M cfg1 es1 cfg2 and trace M cfg2 es2 cfg3*  
**shows** *trace M cfg1 (es1 @ es2) cfg3*  
*<proof>*

We mostly consider traces from the start state to the halting state, for which we introduce the next predicate.

**definition** *traces* :: *machine*  $\Rightarrow$  *tape list*  $\Rightarrow$   $(\text{nat} \times \text{nat})$  *list*  $\Rightarrow$  *tape list*  $\Rightarrow$  *bool* **where**  
*traces M tps1 es tps2*  $\equiv$  *trace M (0, tps1) es (length M, tps2)*

The relation between *traces* and *trace* is like that between *transforms* and *transits*.

**lemma** *tracesI* [*intro*]:

**assumes** *execute M (0, tps1) (length es) = (length M, tps2)*  
**and**  $\bigwedge i. i < \text{length } es \implies \text{fst } (\text{execute } M (0, tps1) i) < \text{length } M$   
**and**  $\bigwedge i. i < \text{length } es \implies$   
*execute M (0, tps1) (Suc i) <#> 0 = fst (es ! i)  $\wedge$*   
*execute M (0, tps1) (Suc i) <#> 1 = snd (es ! i)*  
**shows** *traces M tps1 es tps2*  
*<proof>*

**lemma** *traces-additive*:

**assumes** *trace M (0, tps1) es1 (0, tps2)*  
**and** *traces M tps2 es2 tps3*  
**shows** *traces M tps1 (es1 @ es2) tps3*  
*<proof>*

**lemma** *execute-trace-append*:

**assumes** *trace M1 (0, tps1) es1 (length M1, tps2) (is trace - ?cfg1 - -)*  
**and**  $t \leq \text{length } es1$   
**shows** *execute (M1 @ M2) (0, tps1) t = execute M1 (0, tps1) t*  
*(is execute ?M - - -)*  
*<proof>*

### 5.1.2 Increasing the number of tapes

This is lemma *transforms-append-tapes* adapted for *traces*.

**lemma** *traces-append-tapes*:

**assumes** *turing-machine 2 G M and length tps1 = 2 and traces M tps1 es tps2*  
**shows** *traces (append-tapes 2 (2 + length tps') M) (tps1 @ tps') es (tps2 @ tps')*  
*<proof>*

### 5.1.3 Combining Turing machines

Traces for sequentially composed Turing machines are just concatenated traces of the individual machines.

**lemma** *traces-sequential*:

**assumes** *traces M1 tps1 es1 tps2 and traces M2 tps2 es2 tps3*  
**shows** *traces (M1 ;; M2) tps1 (es1 @ es2) tps3*  
*<proof>*

Next we show how to derive traces for machines created by the *WHILE* operation. If the condition is false, the trace of the loop is the trace for the machine computing the condition plus a singleton trace for the jump.

**lemma** *tm-loop-sem-false-trace*:

**assumes** *traces M1 tps0 es1 tps1*  
**and**  $\neg \text{cond } (\text{read } tps1)$   
**shows** *trace*  
*(WHILE M1 ; cond DO M2 DONE)*  
*(0, tps0)*  
*(es1 @ [(tps1 :# 0, tps1 :# 1)])*  
*(length M1 + length M2 + 2, tps1)*

(is trace ?M - - )  
 ⟨proof⟩

**lemma** *tm-loop-sem-false-traces*:

**assumes** traces M1 tps0 es1 tps1  
**and**  $\neg$  cond (read tps1)  
**and** es = es1 @ [(tps1 :#: 0, tps1 :#: 1)]  
**shows** traces (WHILE M1 ; cond DO M2 DONE) tps0 es tps1  
 ⟨proof⟩

If the loop condition evaluates to true, the trace of one iteration is the concatenation of the traces of the condition machine and the loop body machine with two additional singleton traces for the jumps.

**lemma** *tm-loop-sem-true-traces*:

**assumes** traces M1 tps0 es1 tps1  
**and** traces M2 tps1 es2 tps2  
**and** cond (read tps1)  
**shows** trace  
 (WHILE M1 ; cond DO M2 DONE)  
 (0, tps0)  
 (es1 @ [(tps1 :#: 0, tps1 :#: 1)] @ es2 @ [(tps2 :#: 0, tps2 :#: 1)])  
 (0, tps2)  
 (is trace ?M - ?es -)  
 ⟨proof⟩

**lemma** *tm-loop-sem-true-tracesI*:

**assumes** traces M1 tps0 es1 tps1  
**and** traces M2 tps1 es2 tps2  
**and** cond (read tps1)  
**and** es = es1 @ [(tps1 :#: 0, tps1 :#: 1)] @ es2 @ [(tps2 :#: 0, tps2 :#: 1)]  
**shows** trace (WHILE M1 ; cond DO M2 DONE) (0, tps0) es (0, tps2)  
 ⟨proof⟩

Combining traces for  $m$  iterations of a loop. Typically  $m$  will be the total number of iterations.

**lemma** *tm-loop-trace-simple*:

**fixes** m :: nat  
**and** M :: machine  
**and** tps :: nat  $\Rightarrow$  tape list  
**and** es :: nat  $\Rightarrow$  (nat  $\times$  nat) list  
**assumes**  $\bigwedge i. i < m \implies$  trace M (0, tps i) (es i) (0, tps (Suc i))  
**shows** trace M (0, tps 0) (concat (map es [0.. $m$ ])) (0, tps m)  
 ⟨proof⟩

For simple loops, where we have an upper bound for the length of traces independent of the iteration, there is a trivial upper bound for the length of the trace of  $m$  iterations. This is the only situation we will encounter.

**lemma** *length-concat-le*:

**assumes**  $\bigwedge i. i < m \implies$  length (es i)  $\leq$  b  
**shows** length (concat (map es [0.. $m$ ]))  $\leq$  m \* b  
 ⟨proof⟩

## 5.1.4 Traces for elementary Turing machines

Just like the not necessarily oblivious Turing machines considered so far, our oblivious Turing machines will be built from elementary ones from Section 2.4. In this subsection we show the traces of all the elementary machines we will need.

**lemma** *tm-left-0-traces*:

**assumes** length tps > 1  
**shows** traces  
 (tm-left 0)  
 tps  
 [(tps :#: 0 - 1, tps :#: 1)]  
 (tps[0:=(fst (tps ! 0), snd (tps ! 0) - 1)])

*<proof>*

**lemma** *traces-tm-left-0I*:  
 **assumes**  $\text{length } tps > 1$   
 **and**  $es = [(tps \text{ :\#}: 0 - 1, tps \text{ :\#}: 1)]$   
 **and**  $tps' = (tps[0 := (\text{fst } (tps ! 0), \text{snd } (tps ! 0) - 1)])$   
 **shows** *traces (tm-left 0) tps es tps'*  
 *<proof>*

**lemma** *tm-left-1-traces*:  
 **assumes**  $\text{length } tps > 1$   
 **shows** *traces*  
 (*tm-left 1*)  
  $tps$   
  $[(tps \text{ :\#}: 0, tps \text{ :\#}: 1 - 1)]$   
  $(tps[1 := (\text{fst } (tps ! 1), \text{snd } (tps ! 1) - 1)])$   
 *<proof>*

**lemma** *traces-tm-left-1I*:  
 **assumes**  $\text{length } tps > 1$   
 **and**  $es = [(tps \text{ :\#}: 0, tps \text{ :\#}: 1 - 1)]$   
 **and**  $tps' = (tps[1 := (\text{fst } (tps ! 1), \text{snd } (tps ! 1) - 1)])$   
 **shows** *traces (tm-left 1) tps es tps'*  
 *<proof>*

**lemma** *tm-right-0-traces*:  
 **assumes**  $\text{length } tps > 1$   
 **shows** *traces*  
 (*tm-right 0*)  
  $tps$   
  $[(tps \text{ :\#}: 0 + 1, tps \text{ :\#}: 1)]$   
  $(tps[0 := (\text{fst } (tps ! 0), \text{snd } (tps ! 0) + 1)])$   
 *<proof>*

**lemma** *traces-tm-right-0I*:  
 **assumes**  $\text{length } tps > 1$   
 **and**  $es = [(tps \text{ :\#}: 0 + 1, tps \text{ :\#}: 1)]$   
 **and**  $tps' = (tps[0 := (\text{fst } (tps ! 0), \text{snd } (tps ! 0) + 1)])$   
 **shows** *traces (tm-right 0) tps es tps'*  
 *<proof>*

**lemma** *tm-right-1-traces*:  
 **assumes**  $\text{length } tps > 1$   
 **shows** *traces*  
 (*tm-right 1*)  
  $tps$   
  $[(tps \text{ :\#}: 0, tps \text{ :\#}: 1 + 1)]$   
  $(tps[1 := (\text{fst } (tps ! 1), \text{snd } (tps ! 1) + 1)])$   
 *<proof>*

**lemma** *tm-rtrans-1-traces*:  
 **assumes**  $1 < \text{length } tps$   
 **shows** *traces*  
 (*tm-rtrans 1 f*)  
  $tps$   
  $[(tps \text{ :\#}: 0, tps \text{ :\#}: 1 + 1)]$   
  $(tps[1 := tps ! 1 \mid := f (tps \text{ :\#}: 1) \mid + 1])$   
 *<proof>*

**lemma** *traces-tm-right-1I*:  
 **assumes**  $\text{length } tps > 1$   
 **and**  $es = [(tps \text{ :\#}: 0, tps \text{ :\#}: 1 + 1)]$   
 **and**  $tps' = (tps[1 := (\text{fst } (tps ! 1), \text{snd } (tps ! 1) + 1)])$

**shows traces** (*tm-right* 1) *tps es tps'*  
 ⟨*proof*⟩

**lemma** *traces-tm-rtrans-1I*:

**assumes**  $1 < \text{length } tps$   
**and**  $es = [(tps \text{ :\#} : 0, tps \text{ :\#} : 1 + 1)]$   
**and**  $tps' = (tps[1 := tps ! 1 \mid := \mid f (tps \text{ ::} : 1) \mid + \mid 1])$   
**shows traces** (*tm-rtrans* 1 *f*) *tps es tps'*  
 ⟨*proof*⟩

**lemma** *tm-left-until-1-traces*:

**assumes**  $\text{length } tps > 1$  **and** *begin-tape* *H* ( $tps ! 1$ )  
**shows traces**  
 (*tm-left-until* *H* 1)  
*tps*  
 (*map* ( $\lambda i. (tps \text{ :\#} : 0, i)$ ) (*rev* [ $0..<tps \text{ :\#} : 1$ ]) @  $[(tps \text{ :\#} : 0, 0)]$ )  
 ( $tps[1 := tps ! 1 \mid \# = \mid 0]$ )  
 ⟨*proof*⟩

**lemma** *traces-tm-left-until-1I*:

**assumes**  $\text{length } tps > 1$   
**and** *begin-tape* *H* ( $tps ! 1$ )  
**and**  $es = \text{map } (\lambda i. (tps \text{ :\#} : 0, i)) (\text{rev } [0..<tps \text{ :\#} : 1]) @ [(tps \text{ :\#} : 0, 0)]$   
**and**  $tps' = tps[1 := tps ! 1 \mid \# = \mid 0]$   
**shows traces** (*tm-left-until* *H* 1) *tps es tps'*  
 ⟨*proof*⟩

**lemma** *tm-left-until-0-traces*:

**assumes**  $\text{length } tps > 1$  **and** *begin-tape* *H* ( $tps ! 0$ )  
**shows traces**  
 (*tm-left-until* *H* 0)  
*tps*  
 (*map* ( $\lambda i. (i, tps \text{ :\#} : 1)$ ) (*rev* [ $0..<tps \text{ :\#} : 0$ ]) @  $[(0, tps \text{ :\#} : 1)]$ )  
 ( $tps[0 := tps ! 0 \mid \# = \mid 0]$ )  
 ⟨*proof*⟩

**lemma** *traces-tm-left-until-0I*:

**assumes**  $\text{length } tps > 1$   
**and** *begin-tape* *H* ( $tps ! 0$ )  
**and**  $es = \text{map } (\lambda i. (i, tps \text{ :\#} : 1)) (\text{rev } [0..<tps \text{ :\#} : 0]) @ [(0, tps \text{ :\#} : 1)]$   
**and**  $tps' = tps[0 := tps ! 0 \mid \# = \mid 0]$   
**shows traces** (*tm-left-until* *H* 0) *tps es tps'*  
 ⟨*proof*⟩

**lemma** *tm-start-0-traces*:

**assumes**  $\text{length } tps > 1$  **and** *clean-tape* ( $tps ! 0$ )  
**shows traces**  
 (*tm-start* 0)  
*tps*  
 (*map* ( $\lambda i. (i, tps \text{ :\#} : 1)$ ) (*rev* [ $0..<tps \text{ :\#} : 0$ ]) @  $[(0, tps \text{ :\#} : 1)]$ )  
 ( $tps[0 := tps ! 0 \mid \# = \mid 0]$ )  
 ⟨*proof*⟩

**lemma** *tm-start-1-traces*:

**assumes**  $\text{length } tps > 1$  **and** *clean-tape* ( $tps ! 1$ )  
**shows traces**  
 (*tm-start* 1)  
*tps*  
 (*map* ( $\lambda i. (tps \text{ :\#} : 0, i)$ ) (*rev* [ $0..<tps \text{ :\#} : 1$ ]) @  $[(tps \text{ :\#} : 0, 0)]$ )  
 ( $tps[1 := tps ! 1 \mid \# = \mid 0]$ )  
 ⟨*proof*⟩

**lemma** *traces-tm-start-1I*:

**assumes**  $\text{length } tps > 1$   
**and**  $\text{clean-tape } (tps ! 1)$   
**and**  $es = \text{map } (\lambda i. (tps \text{ :\#} : 0, i)) (\text{rev } [0..<tps \text{ :\#} : 1]) @ [(tps \text{ :\#} : 0, 0)]$   
**and**  $tps' = tps[1 := tps ! 1 \text{ |\#|= } 0]$   
**shows**  $\text{traces } (tm\text{-start } 1) tps \text{ es } tps'$   
 $\langle \text{proof} \rangle$

**lemma**  $tm\text{-cr-0-traces}$ :

**assumes**  $\text{length } tps > 1$  **and**  $\text{clean-tape } (tps ! 0)$   
**shows**  $\text{traces}$   
 $(tm\text{-cr } 0)$   
 $tps$   
 $((\text{map } (\lambda i. (i, tps \text{ :\#} : 1)) (\text{rev } [0..<tps \text{ :\#} : 0]) @ [(0, tps \text{ :\#} : 1)]) @ [(1, tps \text{ :\#} : 1)])$   
 $(tps[0 := tps ! 0 \text{ |\#|= } 1])$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{traces-}tm\text{-cr-0I}$ :

**assumes**  $\text{length } tps > 1$  **and**  $\text{clean-tape } (tps ! 0)$   
**and**  $es = \text{map } (\lambda i. (i, tps \text{ :\#} : 1)) (\text{rev } [0..<tps \text{ :\#} : 0]) @ [(0, tps \text{ :\#} : 1), (1, tps \text{ :\#} : 1)]$   
**and**  $tps' = tps[0 := tps ! 0 \text{ |\#|= } 1]$   
**shows**  $\text{traces } (tm\text{-cr } 0) tps \text{ es } tps'$   
 $\langle \text{proof} \rangle$

**lemma**  $tm\text{-cr-1-traces}$ :

**assumes**  $\text{length } tps > 1$  **and**  $\text{clean-tape } (tps ! 1)$   
**shows**  $\text{traces}$   
 $(tm\text{-cr } 1)$   
 $tps$   
 $((\text{map } (\lambda i. (tps \text{ :\#} : 0, i)) (\text{rev } [0..<tps \text{ :\#} : 1]) @ [(tps \text{ :\#} : 0, 0)]) @ [(tps \text{ :\#} : 0, 1)])$   
 $(tps[1 := tps ! 1 \text{ |\#|= } 1])$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{traces-}tm\text{-cr-1I}$ :

**assumes**  $\text{length } tps > 1$  **and**  $\text{clean-tape } (tps ! 1)$   
**and**  $es = \text{map } (\lambda i. (tps \text{ :\#} : 0, i)) (\text{rev } [0..<tps \text{ :\#} : 1]) @ [(tps \text{ :\#} : 0, 0), (tps \text{ :\#} : 0, 1)]$   
**and**  $tps' = tps[1 := tps ! 1 \text{ |\#|= } 1]$   
**shows**  $\text{traces } (tm\text{-cr } 1) tps \text{ es } tps'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{heads-}tm\text{-trans-until-less}$ :

**assumes**  $j1 < k$   $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{rneigh } (tps ! j1) H n$   
**and**  $t \leq n$   
**shows**  $\text{execute } (tm\text{-trans-until } j1 \ j2 \ H \ f) (0, tps) \ t <\#\> j1 = tps \text{ :\#} : j1 + t$   
**and**  $\text{execute } (tm\text{-trans-until } j1 \ j2 \ H \ f) (0, tps) \ t <\#\> j2 = tps \text{ :\#} : j2 + t$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{heads-}tm\text{-ltrans-until-less}$ :

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{lneigh } (tps ! j1) H n$   
**and**  $t \leq n$   
**and**  $n \leq tps \text{ :\#} : j1$   
**and**  $n \leq tps \text{ :\#} : j2$   
**shows**  $\text{execute } (tm\text{-ltrans-until } j1 \ j2 \ H \ f) (0, tps) \ t <\#\> j1 = tps \text{ :\#} : j1 - t$   
**and**  $\text{execute } (tm\text{-ltrans-until } j1 \ j2 \ H \ f) (0, tps) \ t <\#\> j2 = tps \text{ :\#} : j2 - t$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{heads-}tm\text{-trans-until-less}'$ :

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{rneigh } (tps ! j1) H n$   
**and**  $t \leq n$   
**and**  $j \neq j1$  **and**  $j \neq j2$   
**shows**  $\text{execute } (tm\text{-trans-until } j1 \ j2 \ H \ f) (0, tps) \ t <\#\> j = tps \text{ :\#} : j$

*<proof>*

**lemma** *heads-tm-ltrans-until-less'*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{lneigh } (tps ! j1) H n$   
**and**  $t \leq n$   
**and**  $n \leq tps \text{ :\#} : j1$   
**and**  $n \leq tps \text{ :\#} : j2$   
**and**  $j \neq j1$  **and**  $j \neq j2$   
**shows**  $\text{execute } (tm\text{-}ltrans\text{-}until\ j1\ j2\ H\ f) (0, tps) t <\#\> j = tps \text{ :\#} : j$   
*<proof>*

**lemma** *heads-tm-trans-until*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$  **and**  $\text{rneigh } (tps ! j1) H n$   
**shows**  $\text{execute } (tm\text{-}trans\text{-}until\ j1\ j2\ H\ f) (0, tps) (Suc\ n) <\#\> j1 = tps \text{ :\#} : j1 + n$   
**and**  $\text{execute } (tm\text{-}trans\text{-}until\ j1\ j2\ H\ f) (0, tps) (Suc\ n) <\#\> j2 = tps \text{ :\#} : j2 + n$   
*<proof>*

**lemma** *heads-tm-ltrans-until*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{lneigh } (tps ! j1) H n$   
**and**  $n \leq tps \text{ :\#} : j1$   
**and**  $n \leq tps \text{ :\#} : j2$   
**shows**  $\text{execute } (tm\text{-}ltrans\text{-}until\ j1\ j2\ H\ f) (0, tps) (Suc\ n) <\#\> j1 = tps \text{ :\#} : j1 - n$   
**and**  $\text{execute } (tm\text{-}ltrans\text{-}until\ j1\ j2\ H\ f) (0, tps) (Suc\ n) <\#\> j2 = tps \text{ :\#} : j2 - n$   
*<proof>*

**lemma** *heads-tm-trans-until'*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{rneigh } (tps ! j1) H n$   
**and**  $j \neq j1$  **and**  $j \neq j2$   
**shows**  $\text{execute } (tm\text{-}trans\text{-}until\ j1\ j2\ H\ f) (0, tps) (Suc\ n) <\#\> j = tps \text{ :\#} : j$   
*<proof>*

**lemma** *heads-tm-ltrans-until'*:

**assumes**  $j1 < k$  **and**  $j2 < k$  **and**  $\text{length } tps = k$   
**and**  $\text{lneigh } (tps ! j1) H n$   
**and**  $n \leq tps \text{ :\#} : j1$   
**and**  $n \leq tps \text{ :\#} : j2$   
**and**  $j \neq j1$  **and**  $j \neq j2$   
**shows**  $\text{execute } (tm\text{-}ltrans\text{-}until\ j1\ j2\ H\ f) (0, tps) (Suc\ n) <\#\> j = tps \text{ :\#} : j$   
*<proof>*

**lemma** *traces-tm-trans-until-11*:

**assumes**  $1 < k$  **and**  $\text{length } tps = k$  **and**  $\text{rneigh } (tps ! 1) H n$   
**shows**  $\text{traces } (tm\text{-}trans\text{-}until\ 1\ 1\ H\ f)$   
 $tps$   
 $(\text{map } (\lambda i. (tps \text{ :\#} : 0, tps \text{ :\#} : 1 + Suc\ i)) [0..<n]) @ [(tps \text{ :\#} : 0, tps \text{ :\#} : 1 + n)])$   
 $(tps[1 := \text{transplant } (tps ! 1) (tps ! 1) f\ n])$   
*<proof>*

**lemma** *traces-tm-ltrans-until-11*:

**assumes**  $1 < k$  **and**  $\text{length } tps = k$  **and**  $\text{lneigh } (tps ! 1) H n$  **and**  $n \leq tps \text{ :\#} : 1$   
**shows**  $\text{traces } (tm\text{-}ltrans\text{-}until\ 1\ 1\ H\ f)$   
 $tps$   
 $(\text{map } (\lambda i. (tps \text{ :\#} : 0, tps \text{ :\#} : 1 - Suc\ i)) [0..<n]) @ [(tps \text{ :\#} : 0, tps \text{ :\#} : 1 - n)])$   
 $(tps[1 := \text{ltransplant } (tps ! 1) (tps ! 1) f\ n])$   
*<proof>*

**lemma** *traces-tm-trans-until-01*:

**assumes**  $0 < k$  **and**  $1 < k$  **and**  $\text{length } tps = k$  **and**  $\text{rneigh } (tps ! 0) H n$   
**shows**  $\text{traces } (tm\text{-}trans\text{-}until\ 0\ 1\ H\ f)$   
 $tps$

$(\text{map } (\lambda i. (tps \# : 0 + \text{Suc } i, tps \# : 1 + \text{Suc } i)) [0..<n] @ [(tps \# : 0 + n, tps \# : 1 + n)])$   
 $(tps[0 := tps ! 0 | + | n, 1 := \text{transplant } (tps ! 0) (tps ! 1) f n])$   
 <proof>

**lemma** *traces-tm-trans-until-01I*:

**assumes**  $1 < \text{length } tps$   
**and**  $\text{rneigh } (tps ! 0) H n$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0 + \text{Suc } i, tps \# : 1 + \text{Suc } i)) [0..<n] @ [(tps \# : 0 + n, tps \# : 1 + n)]$   
**and**  $tps' = tps[0 := tps ! 0 | + | n, 1 := \text{transplant } (tps ! 0) (tps ! 1) f n]$   
**shows**  $\text{traces } (tm\text{-trans-until } 0 \ 1 \ H \ f) \ tps \ es \ tps'$   
 <proof>

**lemma** *traces-tm-trans-until-11I*:

**assumes**  $1 < \text{length } tps$   
**and**  $\text{rneigh } (tps ! 1) H n$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0, tps \# : 1 + \text{Suc } i)) [0..<n] @ [(tps \# : 0, tps \# : 1 + n)]$   
**and**  $tps' = tps[1 := \text{transplant } (tps ! 1) (tps ! 1) f n]$   
**shows**  $\text{traces } (tm\text{-trans-until } 1 \ 1 \ H \ f) \ tps \ es \ tps'$   
 <proof>

**lemma** *traces-tm-ltrans-until-11I*:

**assumes**  $1 < \text{length } tps$  **and**  $\forall h < G. f \ h < G$   
**and**  $\text{lneigh } (tps ! 1) H n$   
**and**  $n \leq tps \# : 1$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0, tps \# : 1 - \text{Suc } i)) [0..<n] @ [(tps \# : 0, tps \# : 1 - n)]$   
**and**  $tps' = tps[1 := \text{ltransplant } (tps ! 1) (tps ! 1) f n]$   
**shows**  $\text{traces } (tm\text{-ltrans-until } 1 \ 1 \ H \ f) \ tps \ es \ tps'$   
 <proof>

**lemma** *traces-tm-const-until-01I*:

**assumes**  $1 < \text{length } tps$   
**and**  $\text{rneigh } (tps ! 0) H n$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0 + \text{Suc } i, tps \# : 1 + \text{Suc } i)) [0..<n] @ [(tps \# : 0 + n, tps \# : 1 + n)]$   
**and**  $tps' = tps[0 := tps ! 0 | + | n, 1 := \text{constplant } (tps ! 1) h n]$   
**shows**  $\text{traces } (tm\text{-const-until } 0 \ 1 \ H \ h) \ tps \ es \ tps'$   
 <proof>

**lemma** *traces-tm-const-until-11I*:

**assumes**  $1 < \text{length } tps$  **and**  $h < G$   
**and**  $\text{rneigh } (tps ! 1) H n$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0, tps \# : 1 + \text{Suc } i)) [0..<n] @ [(tps \# : 0, tps \# : 1 + n)]$   
**and**  $tps' = tps[1 := \text{constplant } (tps ! 1) h n]$   
**shows**  $\text{traces } (tm\text{-const-until } 1 \ 1 \ H \ h) \ tps \ es \ tps'$   
 <proof>

**lemma** *traces-tm-cp-until-01I*:

**assumes**  $1 < \text{length } tps$   
**and**  $\text{rneigh } (tps ! 0) H n$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0 + \text{Suc } i, tps \# : 1 + \text{Suc } i)) [0..<n] @ [(tps \# : 0 + n, tps \# : 1 + n)]$   
**and**  $tps' = tps[0 := tps ! 0 | + | n, 1 := \text{implant } (tps ! 0) (tps ! 1) n]$   
**shows**  $\text{traces } (tm\text{-cp-until } 0 \ 1 \ H) \ tps \ es \ tps'$   
 <proof>

**lemma** *traces-tm-cp-until-11I*:

**assumes**  $1 < \text{length } tps$   
**and**  $\text{rneigh } (tps ! 1) H n$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0, tps \# : 1 + \text{Suc } i)) [0..<n] @ [(tps \# : 0, tps \# : 1 + n)]$   
**and**  $tps' = tps[1 := \text{implant } (tps ! 1) (tps ! 1) n]$   
**shows**  $\text{traces } (tm\text{-cp-until } 1 \ 1 \ H) \ tps \ es \ tps'$   
 <proof>

**lemma** *traces-tm-right-until-1I*:

**assumes**  $1 < \text{length } tps$

**and**  $\text{rneigh } (tps ! 1) H n$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0, tps \# : 1 + \text{Suc } i)) [0..<n] @ [(tps \# : 0, tps \# : 1 + n)]$   
**and**  $tps' = tps[1 := (tps ! 1) | + | n]$   
**shows**  $\text{traces } (tm\text{-right-until } 1 H) tps es tps'$   
 $\langle \text{proof} \rangle$

**lemma** *execute-tm-write*:

**shows**  $\text{execute } (tm\text{-write } j h) (0, tps) 1 = (1, tps[j := tps ! j | := | h])$   
 $\langle \text{proof} \rangle$

**lemma** *traces-tm-writeI*:

**assumes**  $j > 0$  **and**  $j < \text{length } tps$   
**and**  $es = [(tps \# : 0, tps \# : 1)]$   
**and**  $tps' = tps[j := tps ! j | := | h]$   
**shows**  $\text{traces } (tm\text{-write } j h) tps es tps'$   
 $\langle \text{proof} \rangle$

**corollary** *traces-tm-write-1I*:

**assumes**  $1 < \text{length } tps$   
**and**  $es = [(tps \# : 0, tps \# : 1)]$   
**and**  $tps' = tps[1 := tps ! 1 | := | h]$   
**shows**  $\text{traces } (tm\text{-write } 1 h) tps es tps'$   
 $\langle \text{proof} \rangle$

**corollary** *traces-tm-write-ge2I*:

**assumes**  $j \geq 2$   
**and**  $j < \text{length } tps$   
**and**  $es = [(tps \# : 0, tps \# : 1)]$   
**and**  $tps' = tps[j := tps ! j | := | h]$   
**shows**  $\text{traces } (tm\text{-write } j h) tps es tps'$   
 $\langle \text{proof} \rangle$

**lemma** *execute-tm-write-manyI*:

**assumes**  $0 \notin J$  **and**  $\forall j \in J. j < k$  **and**  $k \geq 2$  **and**  $\text{length } tps = k$   
**and**  $\text{length } tps' = k$   
**and**  $\bigwedge j. j \in J \implies tps' ! j = tps ! j | := | h$   
**and**  $\bigwedge j. j < k \implies j \notin J \implies tps' ! j = tps ! j$   
**shows**  $\text{execute } (tm\text{-write-many } J h) (0, tps) 1 = (1, tps')$   
 $\langle \text{proof} \rangle$

**lemma** *traces-tm-write-manyI*:

**assumes**  $0 \notin J$  **and**  $\forall j \in J. j < k$  **and**  $k \geq 2$  **and**  $\text{length } tps = k$   
**and**  $\text{length } tps' = k$   
**and**  $\bigwedge j. j \in J \implies tps' ! j = tps ! j | := | h$   
**and**  $\bigwedge j. j < k \implies j \notin J \implies tps' ! j = tps ! j$   
**and**  $es = [(tps \# : 0, tps \# : 1)]$   
**shows**  $\text{traces } (tm\text{-write-many } J h) tps es tps'$   
 $\langle \text{proof} \rangle$

**lemma** *traces-tm-write-repeat-1I*:

**assumes**  $1 < \text{length } tps$   
**and**  $es = \text{map } (\lambda i. (tps \# : 0, tps \# : 1 + \text{Suc } i)) [0..<m]$   
**and**  $tps' = tps[1 := \text{overwrite } (tps ! 1) h m]$   
**shows**  $\text{traces } (tm\text{-write-repeat } 1 h m) tps es tps'$   
 $\langle \text{proof} \rangle$

### 5.1.5 Memorizing in states

We need some results for the traces of “cartesian” machines used for the memorizing-in-states technique introduced in Section 2.5.

**lemma** *cartesian-trace*:

**assumes** *turing-machine*  $(\text{Suc } k) G M$   
**and** *immobile*  $M k (\text{Suc } k)$

```

and  $M' = \text{cartesian } M \ G$ 
and  $k \geq 2$ 
and  $\forall i < \text{length } zs. zs ! i < G$ 
and  $\text{trace } M \ (\text{start-config } (\text{Suc } k) \ zs) \ \text{es } \text{cfg}$ 
shows  $\text{trace } M' \ (\text{start-config } k \ zs) \ \text{es } (\text{squish } G \ (\text{length } M) \ \text{cfg})$ 
<proof>

```

**lemma cartesian-traces:**

```

assumes  $\text{turing-machine } (\text{Suc } k) \ G \ M$ 
and  $\text{immobile } M \ k \ (\text{Suc } k)$ 
and  $M' = \text{cartesian } M \ G$ 
and  $k \geq 2$ 
and  $\forall i < \text{length } zs. zs ! i < G$ 
and  $\text{traces } M \ (\text{snd } (\text{start-config } (\text{Suc } k) \ zs)) \ \text{es } \text{tps}$ 
shows  $\text{traces } M' \ (\text{snd } (\text{start-config } k \ zs)) \ \text{es } (\text{butlast } \text{tps})$ 
<proof>

```

**lemma traces-tapes-length:**

```

assumes  $\text{turing-machine } k \ G \ M$ 
and  $\text{length } \text{tps} = k$ 
and  $\text{traces } M \ \text{tps} \ \text{es } \text{tps}'$ 
shows  $\text{length } \text{tps}' = k$ 
<proof>

```

**lemma icartesian:**

```

assumes  $\text{turing-machine } (k + 2) \ G \ M$ 
and  $\bigwedge j. j < k \implies \text{immobile } M \ (j + 2) \ (k + 2)$ 
and  $\forall i < \text{length } zs. zs ! i < G$ 
and  $\text{traces } M \ (\text{snd } (\text{start-config } (k + 2) \ zs)) \ \text{es } \text{tps}$ 
shows  $\text{traces } (\text{icartesian } k \ M \ G) \ (\text{snd } (\text{start-config } 2 \ zs)) \ \text{es } (\text{take } 2 \ \text{tps})$ 
<proof>

```

**end**

## 5.2 Constructing polynomials in polynomial time

**theory Oblivious-Polynomial**

**imports Oblivious**

**begin**

Our current goal is to simulate a polynomial time multi-tape TM on a two-tape oblivious TM in polynomial time. To help with the obliviousness we first want to mark on the simulator's output tape a space that is at least as large as the space the simulated TM uses on any of its tapes but that still is only polynomial in size. In this section we construct oblivious Turing machines for that.

An upper bound for the size this space is provided by the simulated TM's running time, which by assumption is polynomially bounded. So for our purposes any polynomially bounded function that bounds the running time will do.

In this section we devise a family of two-tape oblivious TMs that contains for every polynomial degree  $d \geq 1$  a TM that writes  $\mathbf{1}^{p(n)}$  to the output tape for some polynomial  $p$  with  $p(n) \geq n^d$ , where  $n$  is the length of the input to the TM. Together with a TM that appends a constant number  $c$  of ones we get a family of TMs, parameterized by  $c$  and  $d$ , that runs in polynomial time and writes more than  $c + n^d$  symbols  $\mathbf{1}$  to the work tape.

This meets our goal for this section because every polynomially bounded function is bounded by a function of the form  $n \mapsto c + n^d$  for some  $c, d \in \mathbb{N}$ .

The TMs in the family are built from three building block TMs. The first TM initializes its output tape with  $\mathbf{1}^n$  where  $n$  is the length of the input. The second TM multiplies the number of symbols on the output tape by (roughly) the length of the input, turning a sequence  $\mathbf{1}^m$  into (roughly)  $\mathbf{1}^{mn}$  for arbitrary  $m$ . The third TM appends  $\mathbf{1}^c$  for some constant  $c$ . By repeating the second TM we can achieve arbitrarily high polynomial degrees.

All three TMs do essentially only one thing with the input, namely copying it to the output tape while

mapping all symbols to **1**, which is an operation that depends only on the length of the input. Therefore all three TMs are oblivious, and combining them also yields an oblivious TM.

The Turing machines require one extra symbol beyond the four default symbols, but work for all alphabet sizes  $G \geq 5$ .

```

locale turing-machine-poly =
  fixes G :: nat
  assumes G: G  $\geq$  5
begin

```

```

lemma G-ge4 [simp]: G  $\geq$  4
  <proof>

```

```

abbreviation tps-ones :: symbol list  $\Rightarrow$  nat  $\Rightarrow$  tape list where
  tps-ones zs m  $\equiv$ 
    [([zs], 1),
     ( $\lambda i$ . if i = 0 then  $\triangleright$  else if i  $\leq$  m then 1 else  $\square$ , 1)]

```

### 5.2.1 Initializing the output tape

The first building block is a TM that “copies” the input to the output tape, thereby replacing every symbol by the symbol **1**.

```

definition tmA :: machine where
  tmA  $\equiv$ 
    tm-right 0 ;; tm-right 1 ;; tm-const-until 0 1 { $\square$ } 1 ;; tm-cr 0 ;; tm-cr 1

```

```

lemma tmA-tm: turing-machine 2 G tmA
  <proof>

```

```

definition tm1  $\equiv$  tm-right 0
definition tm2  $\equiv$  tm1 ;; tm-right 1
definition tm3  $\equiv$  tm2 ;; tm-const-until 0 1 { $\square$ } 1
definition tm4  $\equiv$  tm3 ;; tm-cr 0
definition tm5  $\equiv$  tm4 ;; tm-cr 1

```

```

lemma tm5-eq-tmA: tm5 = tmA
  <proof>

```

```

definition tps0 :: symbol list  $\Rightarrow$  tape list where
  tps0 zs  $\equiv$ 
    [([zs], 0),
     ( $\lambda i$ . if i = 0 then  $\triangleright$  else  $\square$ , 0)]

```

```

lemma length-tps0 [simp]: length (tps0 n) = 2
  <proof>

```

```

definition tps1 :: symbol list  $\Rightarrow$  tape list where
  tps1 zs  $\equiv$ 
    [([zs], 1),
     ( $\lambda i$ . if i = 0 then  $\triangleright$  else  $\square$ , 0)]

```

```

definition es1 :: (nat  $\times$  nat) list where
  es1  $\equiv$  [(1, 0)]

```

```

lemma tm1: traces tm1 (tps0 zs) es1 (tps1 zs)
  <proof>

```

```

definition tps2 :: symbol list  $\Rightarrow$  tape list where
  tps2 zs  $\equiv$ 
    [([zs], 1),
     ( $\lambda i$ . if i = 0 then  $\triangleright$  else  $\square$ , 1)]

```

**definition**  $es2 :: (nat \times nat) list$  **where**  
 $es2 \equiv es1 @ [(1, 1)]$

**lemma**  $length-es2: length\ es2 = 2$   
 $\langle proof \rangle$

**lemma**  $tm2: traces\ tm2\ (tps0\ zs)\ es2\ (tps2\ zs)$   
 $\langle proof \rangle$

**definition**  $tps3 :: symbol\ list \Rightarrow tape\ list$  **where**  
 $tps3\ zs \equiv$   
 $[(\lfloor zs \rfloor, length\ zs + 1),$   
 $(\lambda i. if\ i = 0\ then\ \triangleright\ else\ if\ i \leq length\ zs\ then\ \mathbf{1}\ else\ \square, length\ zs + 1)]$

**definition**  $es23 :: nat \Rightarrow (nat \times nat) list$  **where**  
 $es23\ n \equiv map\ (\lambda i. (i + 2, i + 2))\ [0..<n] @ [(n + 1, n + 1)]$

**definition**  $es3 :: nat \Rightarrow (nat \times nat) list$  **where**  
 $es3\ n \equiv es2 @ (es23\ n)$

**lemma**  $length-es3: length\ (es3\ n) = n + 3$   
 $\langle proof \rangle$

**lemma**  $tm3:$   
**assumes**  $proper-symbols\ zs$   
**shows**  $traces\ tm3\ (tps0\ zs)\ (es3\ (length\ zs))\ (tps3\ zs)$   
 $\langle proof \rangle$

**definition**  $tps4 :: symbol\ list \Rightarrow tape\ list$  **where**  
 $tps4\ zs \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lambda i. if\ i = 0\ then\ \triangleright\ else\ if\ i \leq length\ zs\ then\ \mathbf{1}\ else\ \square, length\ zs + 1)]$

**definition**  $es34 :: nat \Rightarrow (nat \times nat) list$  **where**  
 $es34\ n \equiv map\ (\lambda i. (i, n + 1))\ (rev\ [0..<n + 1]) @ [(0, n + 1)] @ [(1, n + 1)]$

**definition**  $es4 :: nat \Rightarrow (nat \times nat) list$  **where**  
 $es4\ n \equiv es3\ n @ es34\ n$

**lemma**  $length-es4: length\ (es4\ n) = 2 * n + 6$   
 $\langle proof \rangle$

**lemma**  $tm4:$   
**assumes**  $proper-symbols\ zs$   
**shows**  $traces\ tm4\ (tps0\ zs)\ (es4\ (length\ zs))\ (tps4\ zs)$   
 $\langle proof \rangle$

**definition**  $tps5 :: symbol\ list \Rightarrow nat \Rightarrow tape\ list$  **where**  
 $tps5\ zs\ m \equiv tps-ones\ zs\ m$

**definition**  $es45 :: nat \Rightarrow (nat \times nat) list$  **where**  
 $es45\ n \equiv map\ (\lambda i. (1, i))\ (rev\ [0..<n + 1]) @ [(1, 0)] @ [(1, 1)]$

**definition**  $es5 :: nat \Rightarrow (nat \times nat) list$  **where**  
 $es5\ n \equiv es4\ n @ es45\ n$

**lemma**  $length-es5: length\ (es5\ n) = 3 * n + 9$   
 $\langle proof \rangle$

**lemma**  $tm5:$   
**assumes**  $proper-symbols\ zs$   
**shows**  $traces\ tm5\ (tps0\ zs)\ (es5\ (length\ zs))\ (tps5\ zs\ (length\ zs))$   
 $\langle proof \rangle$

**corollary** *tmA*:

**assumes** *proper-symbols zs*

**shows** *traces tmA (tps0 zs) (es5 (length zs)) (tps-ones zs (length zs))*

*<proof>*

**lemma** *length-tps-ones [simp]*: *length (tps-ones zs m) = 2*

*<proof>*

## 5.2.2 Multiplying by the input length

The next Turing machine turns a symbol sequence  $\mathbf{1}^m$  on its output tape into  $\mathbf{1}^{m+1+mn}$  where  $n$  is the length of the input.

The TM first appends to the output tape symbols a  $|$  symbol. Then it performs a loop with  $m$  iterations. In each iteration it replaces a  $\mathbf{1}$  before the  $|$  by  $\mathbf{0}$  in order to count the iterations. Then it copies (replacing each symbol by  $\mathbf{1}$ ) the input after the output tape symbols. After the loop the output tape contains  $\mathbf{0}^m|\mathbf{1}^{mn}$ . Finally the  $|$  and  $\mathbf{0}$  symbols are replaced by  $\mathbf{1}$  symbols, yielding the desired result.

**definition** *tmB* :: *machine* **where**

*tmB*  $\equiv$

*tm-right-until* 1  $\{\square\}$  ;;

*tm-write* 1  $|$  ;;

*tm-cr* 1 ;;

*WHILE* *tm-right-until* 1  $\{\mathbf{1}, |\}$  ;  $\lambda rs. rs ! 1 = \mathbf{1}$  *DO*

*tm-write* 1  $\mathbf{0}$  ;;

*tm-right-until* 1  $\{0\}$  ;;

*tm-const-until* 0 1  $\{\square\}$   $\mathbf{1}$  ;;

*tm-cr* 0 ;;

*tm-cr* 1

*DONE* ;;

*tm-write* 1  $\mathbf{1}$  ;;

*tm-cr* 1 ;;

*tm-const-until* 1 1  $\{\mathbf{1}\}$   $\mathbf{1}$  ;;

*tm-cr* 1

**lemma** *tmB-tm*: *turing-machine 2 G tmB*

*<proof>*

**definition** *tmB1*  $\equiv$  *tm-right-until* 1  $\{\square\}$

**definition** *tmB2*  $\equiv$  *tmB1* ;; *tm-write* 1  $|$

**definition** *tmB3*  $\equiv$  *tmB2* ;; *tm-cr* 1

**definition** *tmK1*  $\equiv$  *tm-right-until* 1  $\{\mathbf{1}, |\}$

**definition** *tmL1*  $\equiv$  *tm-write* 1  $\mathbf{0}$

**definition** *tmL2*  $\equiv$  *tmL1* ;; *tm-right-until* 1  $\{\square\}$

**definition** *tmL3*  $\equiv$  *tmL2* ;; *tm-const-until* 0 1  $\{\square\}$   $\mathbf{1}$

**definition** *tmL4*  $\equiv$  *tmL3* ;; *tm-cr* 0

**definition** *tmL5*  $\equiv$  *tmL4* ;; *tm-cr* 1

**definition** *tmLoop*  $\equiv$  *WHILE* *tmK1* ;  $\lambda rs. rs ! 1 = \mathbf{1}$  *DO* *tmL5* *DONE*

**definition** *tmB4*  $\equiv$  *tmB3* ;; *tmLoop*

**definition** *tmB5*  $\equiv$  *tmB4* ;; *tm-write* 1  $\mathbf{1}$

**definition** *tmB6*  $\equiv$  *tmB5* ;; *tm-cr* 1

**definition** *tmB7*  $\equiv$  *tmB6* ;; *tm-const-until* 1 1  $\{\mathbf{1}\}$   $\mathbf{1}$

**definition** *tmB8*  $\equiv$  *tmB7* ;; *tm-cr* 1

**lemma** *tmB-eq-tmB8*: *tmB = tmB8*

*<proof>*

**definition** *tpsB1* :: *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape list* **where**

*tpsB1* *zs* *m*  $\equiv$

$[(\lfloor zs \rfloor, 1),$

$(\lambda i. \text{if } i = 0 \text{ then } \triangleright \text{ else if } i \leq m \text{ then } \mathbf{1} \text{ else } \square, m + 1)]$

**definition** *esB1* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$   $(\text{nat} \times \text{nat})$  *list* **where**

$esB1\ n\ m \equiv \text{map } (\lambda i. (1, 1 + \text{Suc } i)) [0..<m] @ [(1, 1 + m)]$

**lemma** *length-esB1*:  $\text{length } (esB1\ n\ m) = m + 1$   
 ⟨proof⟩

**lemma** *tmB1*:

**assumes** *proper-symbols zs*

**shows** *traces tmB1 (tps-ones zs m) (esB1 (length zs) m) (tpsB1 zs m)*

⟨proof⟩

**definition** *tpsB2* :: *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape list* **where**

$tpsB2\ zs\ m \equiv$

$[(\lfloor zs \rfloor, 1),$

$(\lambda i. \text{if } i = 0 \text{ then } \triangleright \text{ else if } i \leq m \text{ then } \mathbf{1} \text{ else if } i = m + 1 \text{ then } | \text{ else } \square, m + 1)]$

**definition** *esB12* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  (*nat*  $\times$  *nat*) *list* **where**

$esB12\ n\ m \equiv [(1, m + 1)]$

**definition** *esB2* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  (*nat*  $\times$  *nat*) *list* **where**

$esB2\ n\ m \equiv esB1\ n\ m @ esB12\ n\ m$

**lemma** *length-esB2*:  $\text{length } (esB2\ n\ m) = m + 2$

⟨proof⟩

**lemma** *tmB2*:

**assumes** *proper-symbols zs*

**shows** *traces tmB2 (tps-ones zs m) (esB2 (length zs) m) (tpsB2 zs m)*

⟨proof⟩

**definition** *tpsB3* :: *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape list* **where**

$tpsB3\ zs\ m \equiv$

$[(\lfloor zs \rfloor, 1),$

$(\lambda i. \text{if } i = 0 \text{ then } \triangleright \text{ else if } i \leq m \text{ then } \mathbf{1} \text{ else if } i = m + 1 \text{ then } | \text{ else } 0, 1)]$

**definition** *esB23* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  (*nat*  $\times$  *nat*) *list* **where**

$esB23\ n\ m \equiv \text{map } (\text{Pair } 1) (\text{rev } [0..<m + 1]) @ [(1, 0), (1, 1)]$

**definition** *esB3* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  (*nat*  $\times$  *nat*) *list* **where**

$esB3\ n\ m \equiv esB2\ n\ m @ esB23\ n\ m$

**lemma** *length-esB3*:  $\text{length } (esB3\ n\ m) = 2 * m + 5$

⟨proof⟩

**lemma** *tmB3*:

**assumes** *proper-symbols zs*

**shows** *traces tmB3 (tps-ones zs m) (esB3 (length zs) m) (tpsB3 zs m)*

⟨proof⟩

**definition** *tpsK0* :: *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape list* **where**

$tpsK0\ zs\ m\ i \equiv$

$[(\lfloor zs \rfloor, 1),$

$(\lambda x. \text{if } x = 0 \text{ then } \triangleright$

$\text{ else if } x \leq i \text{ then } \mathbf{0}$

$\text{ else if } x \leq m \text{ then } \mathbf{1}$

$\text{ else if } x = m + 1 \text{ then } |$

$\text{ else if } x \leq m + 1 + i * \text{length } zs \text{ then } \mathbf{1}$

$\text{ else } 0,$

$1)]$

**lemma** *tpsK0-eq-tpsB3*:  $tpsK0\ zs\ m\ 0 = tpsB3\ zs\ m$

⟨proof⟩

**definition** *tpsK1* :: *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape list* **where**

*tpsK1*  $zs\ m\ i \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq i \text{ then } \mathbf{0}$   
 $\quad \text{else if } x \leq m \text{ then } \mathbf{1}$   
 $\quad \text{else if } x = m + 1 \text{ then } |$   
 $\quad \text{else if } x \leq m + 1 + i * \text{length } zs \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $\quad i + 1)]$

**definition** *esK1*  $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esK1\ n\ m\ i \equiv \text{map } (\lambda i. (1, 1 + \text{Suc } i)) [0..<i] @ [(1, i + 1)]$

**lemma** *length-esK1*:  $\text{length } (esK1\ n\ m\ i) = i + 1$   
 $\langle \text{proof} \rangle$

**lemma** *tmK1*:  
**assumes** *proper-symbols*  $zs$  **and**  $i < m$   
**shows** *traces* *tmK1*  $(tpsK0\ zs\ m\ i) (esK1\ (\text{length } zs)\ m\ i) (tpsK1\ zs\ m\ i)$   
 $\langle \text{proof} \rangle$

**definition** *tpsL1*  $:: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{tape list where}$   
 $tpsL1\ zs\ m\ i \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq i + 1 \text{ then } \mathbf{0}$   
 $\quad \text{else if } x \leq m \text{ then } \mathbf{1}$   
 $\quad \text{else if } x = m + 1 \text{ then } |$   
 $\quad \text{else if } x \leq m + 1 + i * \text{length } zs \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $\quad i + 1)]$

**definition** *esL1*  $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esL1\ n\ m\ i \equiv [(1, i + 1)]$

**lemma** *tmL1*:  
**assumes** *proper-symbols*  $zs$   
**shows** *traces* *tmL1*  $(tpsK1\ zs\ m\ i) (esL1\ (\text{length } zs)\ m\ i) (tpsL1\ zs\ m\ i)$   
 $\langle \text{proof} \rangle$

**definition** *tpsL2*  $:: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{tape list where}$   
 $tpsL2\ zs\ m\ i \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq i + 1 \text{ then } \mathbf{0}$   
 $\quad \text{else if } x \leq m \text{ then } \mathbf{1}$   
 $\quad \text{else if } x = m + 1 \text{ then } |$   
 $\quad \text{else if } x \leq m + 1 + i * \text{length } zs \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $\quad m + 2 + i * \text{length } zs)]$

**definition** *esL12*  $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esL12\ n\ m\ i \equiv$   
 $\text{map } (\lambda j. (1, i + 1 + \text{Suc } j)) [0..<m + 2 + i * n - (i + 1)] @$   
 $[(1, i + 1 + (m + 2 + i * n - (i + 1)))]$

**definition** *esL2*  $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esL2\ n\ m\ i \equiv esL1\ n\ m\ i @ esL12\ n\ m\ i$

**lemma** *length-esL2*:  $i < m \implies \text{length } (esL2\ n\ m\ i) = 3 + m - i + i * n$   
 $\langle \text{proof} \rangle$

A simplified upper bound for the running time:

**corollary**  $\text{length-esL2}' : i < m \implies \text{length} (\text{esL2 } n \ m \ i) \leq 3 + m + i * n$   
 ⟨proof⟩

**lemma**  $\text{tmL2}$ :

**assumes** proper-symbols  $zs$  **and**  $i < m$   
**shows** traces  $\text{tmL2}$  ( $\text{tpsK1 } zs \ m \ i$ ) ( $\text{esL2} (\text{length } zs) \ m \ i$ ) ( $\text{tpsL2 } zs \ m \ i$ )  
 ⟨proof⟩

**definition**  $\text{tpsL3} :: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL3 } zs \ m \ i \equiv$   
 $[[[zs], \text{length } zs + 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq i + 1 \text{ then } \mathbf{0}$   
 $\quad \text{else if } x \leq m \text{ then } \mathbf{1}$   
 $\quad \text{else if } x = m + 1 \text{ then } |$   
 $\quad \text{else if } x \leq m + 1 + (i + 1) * \text{length } zs \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $m + 2 + (i + 1) * \text{length } zs]]$

**definition**  $\text{esL23} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})$  list **where**

$\text{esL23 } n \ m \ i \equiv$   
 $\text{map } (\lambda j. (1 + \text{Suc } j, m + 2 + i * n + \text{Suc } j)) [0..<n] @ [(1 + n, m + 2 + i * n + n)]$

**definition**  $\text{esL3} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})$  list **where**

$\text{esL3 } n \ m \ i \equiv \text{esL2 } n \ m \ i @ \text{esL23 } n \ m \ i$

**lemma**  $\text{length-esL3} : i < m \implies \text{length} (\text{esL3 } n \ m \ i) \leq 4 + m + (i + 1) * n$   
 ⟨proof⟩

**lemma**  $\text{tmL3}$ :

**assumes** proper-symbols  $zs$  **and**  $i < m$   
**shows** traces  $\text{tmL3}$  ( $\text{tpsK1 } zs \ m \ i$ ) ( $\text{esL3} (\text{length } zs) \ m \ i$ ) ( $\text{tpsL3 } zs \ m \ i$ )  
 ⟨proof⟩

**definition**  $\text{tpsL4} :: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL4 } zs \ m \ i \equiv$   
 $[[[zs], 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq i + 1 \text{ then } \mathbf{0}$   
 $\quad \text{else if } x \leq m \text{ then } \mathbf{1}$   
 $\quad \text{else if } x = m + 1 \text{ then } |$   
 $\quad \text{else if } x \leq m + 1 + (i + 1) * \text{length } zs \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $m + 2 + (i + 1) * \text{length } zs]]$

**definition**  $\text{esL34} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})$  list **where**

$\text{esL34 } n \ m \ i \equiv$   
 $\text{map } (\lambda j. (j, m + 2 + (i + 1) * n)) (\text{rev } [0..<n + 1]) @ [(0, m + 2 + (i + 1) * n), (1, m + 2 + (i + 1) * n)]$

**lemma**  $\text{length-esL34} : i < m \implies \text{length} (\text{esL34 } n \ m \ i) = n + 3$   
 ⟨proof⟩

**definition**  $\text{esL4} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})$  list **where**

$\text{esL4 } n \ m \ i \equiv \text{esL3 } n \ m \ i @ \text{esL34 } n \ m \ i$

**lemma**  $\text{length-esL4} : i < m \implies \text{length} (\text{esL4 } n \ m \ i) \leq 7 + m + (i + 2) * n$   
 ⟨proof⟩

**lemma**  $\text{tmL4}$ :

**assumes** proper-symbols  $zs$  **and**  $i < m$   
**shows** traces  $\text{tmL4}$  ( $\text{tpsK1 } zs \ m \ i$ ) ( $\text{esL4} (\text{length } zs) \ m \ i$ ) ( $\text{tpsL4 } zs \ m \ i$ )  
 ⟨proof⟩

**definition**  $tpsL5 :: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsL5\ zs\ m\ i \equiv$   
 $[[[zs], 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq i + 1 \text{ then } \mathbf{0}$   
 $\quad \text{else if } x \leq m \text{ then } \mathbf{1}$   
 $\quad \text{else if } x = m + 1 \text{ then } |$   
 $\quad \text{else if } x \leq m + 1 + (i + 1) * \text{length } zs \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $1)]]$

**definition**  $esL45 :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})$  *list* **where**

$esL45\ n\ m\ i \equiv \text{map } (\text{Pair } 1) (\text{rev } [0..<m + 2 + (i + 1) * n]) @ [(1, 0), (1, 1)]$

**definition**  $esL5 :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})$  *list* **where**

$esL5\ n\ m\ i \equiv esL4\ n\ m\ i @ esL45\ n\ m\ i$

**lemma**  $\text{length-esL5}: i < m \implies \text{length } (esL5\ n\ m\ i) \leq 11 + 2 * m + (2 * i + 3) * n$   
*<proof>*

**lemma**  $tmL5:$

**assumes** *proper-symbols*  $zs$  **and**  $i < m$   
**shows**  $\text{traces } tmL5\ (tpsK1\ zs\ m\ i)\ (esL5\ (\text{length } zs)\ m\ i)\ (tpsL5\ zs\ m\ i)$   
*<proof>*

**definition**  $esLoop\text{-do} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})$  *list* **where**

$esLoop\text{-do}\ n\ m\ i \equiv esK1\ n\ m\ i @ [(1, i + 1)] @ esL5\ n\ m\ i @ [(1, 1)]$

Using  $i < m$  we get this upper bound for the running time of an iteration independent of  $i$ .

**lemma**  $\text{length-esLoop-do}: i < m \implies \text{length } (esLoop\text{-do}\ n\ m\ i) \leq 14 + 3 * m + (2 * m + 3) * n$   
*<proof>*

**lemma**  $tmLoop\text{-do}:$

**assumes** *proper-symbols*  $zs$  **and**  $i < m$   
**shows**  $\text{trace } tmLoop\ (0, tpsK0\ zs\ m\ i)\ (esLoop\text{-do}\ (\text{length } zs)\ m\ i)\ (0, tpsL5\ zs\ m\ i)$   
*<proof>*

**lemma**  $tpsL5\text{-eq-tpsK0}:$

**assumes** *proper-symbols*  $zs$  **and**  $i < m$   
**shows**  $tpsL5\ zs\ m\ i = tpsK0\ zs\ m\ (Suc\ i)$   
*<proof>*

**lemma**  $tmLoop\text{-iteration}:$

**assumes** *proper-symbols*  $zs$  **and**  $i < m$   
**shows**  $\text{trace } tmLoop\ (0, tpsK0\ zs\ m\ i)\ (esLoop\text{-do}\ (\text{length } zs)\ m\ i)\ (0, tpsK0\ zs\ m\ (Suc\ i))$   
*<proof>*

**definition**  $esLoop\text{-done} :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})$  *list* **where**

$esLoop\text{-done}\ n\ m \equiv \text{concat } (\text{map } (esLoop\text{-do}\ n\ m)\ [0..<m])$

**lemma**  $tmLoop\text{-done}:$

**assumes** *proper-symbols*  $zs$   
**shows**  $\text{trace } tmLoop\ (0, tpsK0\ zs\ m\ 0)\ (esLoop\text{-done}\ (\text{length } zs)\ m)\ (0, tpsK0\ zs\ m\ m)$   
*<proof>*

**lemma**  $\text{length-esLoop-done}: \text{length } (esLoop\text{-done}\ n\ m) \leq m * (14 + 3 * m + (2 * m + 3) * n)$   
*<proof>*

**definition**  $tpsK\text{-break} :: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsK\text{-break}\ zs\ m \equiv$   
 $[[[zs], 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$

else if  $x \leq m$  then **0**  
 else if  $x = m + 1$  then |  
 else if  $x \leq m + 1 + m * \text{length } zs$  then **1**  
 else 0,  
 m + 1]]

**definition** *esK-break* :: nat ⇒ nat ⇒ (nat × nat) list **where**  
*esK-break* n m ≡ map (λi. (1, 1 + Suc i)) [0.. $m$ ] @ [(1, 1 + m)]

**lemma** *length-esK-break*: length (*esK-break* n m) = m + 1  
 ⟨proof⟩

**lemma** *tmK1-break*:  
 assumes *proper-symbols* zs  
 shows traces *tmK1* (*tpsK0* zs m m) (*esK-break* (length zs) m) (*tpsK-break* zs m)  
 ⟨proof⟩

**definition** *esLoop-break* :: nat ⇒ nat ⇒ (nat × nat) list **where**  
*esLoop-break* n m ≡ *esK-break* n m @ [(1, m + 1)]

**lemma** *length-esLoop-break*: length (*esLoop-break* n m) = m + 2  
 ⟨proof⟩

**lemma** *tmLoop-break*:  
 assumes *proper-symbols* zs  
 shows traces *tmLoop* (*tpsK0* zs m m) (*esLoop-break* (length zs) m) (*tpsK-break* zs m)  
 ⟨proof⟩

**definition** *esLoop* :: nat ⇒ nat ⇒ (nat × nat) list **where**  
*esLoop* n m ≡ *esLoop-done* n m @ *esLoop-break* n m

**lemma** *length-esLoop*: length (*esLoop* n m) ≤ m \* (14 + 3 \* m + (2 \* m + 3) \* n) + m + 2  
 ⟨proof⟩

**lemma** *length-esLoop'*: length (*esLoop* n m) ≤ 2 + 18 \* m \* m + 5 \* m \* m \* n  
 ⟨proof⟩

**lemma** *tmLoop*:  
 assumes *proper-symbols* zs  
 shows traces *tmLoop* (*tpsK0* zs m 0) (*esLoop* (length zs) m) (*tpsK-break* zs m)  
 ⟨proof⟩

**lemma** *tmLoop'*:  
 assumes *proper-symbols* zs  
 shows traces *tmLoop* (*tpsB3* zs m) (*esLoop* (length zs) m) (*tpsK-break* zs m)  
 ⟨proof⟩

**definition** *esB4* :: nat ⇒ nat ⇒ (nat × nat) list **where**  
*esB4* n m ≡ *esB3* n m @ *esLoop* n m

**lemma** *length-esB4*: length (*esB4* n m) ≤ 7 + 20 \* m \* m + 5 \* m \* m \* n  
 ⟨proof⟩

**lemma** *tmB4*:  
 assumes *proper-symbols* zs  
 shows traces *tmB4* (*tps-ones* zs m) (*esB4* (length zs) m) (*tpsK-break* zs m)  
 ⟨proof⟩

**definition** *tpsB5* :: symbol list ⇒ nat ⇒ tape list **where**  
*tpsB5* zs m ≡  
 [(⌊zs⌋, 1),  
 (λx. if x = 0 then ▷  
 else if x ≤ m then **0**

else if  $x \leq m + 1 + m * \text{length } zs$  then **1**  
 else 0,  
 m + 1)]

**definition**  $esB5 :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esB5 \ n \ m \equiv esB4 \ n \ m \ @ \ [(1, m + 1)]$

**lemma**  $\text{length-esB5}$ :  $\text{length} (esB5 \ n \ m) \leq 8 + 20 * m * m + 5 * m * m * n$   
 ⟨proof⟩

**lemma**  $tmB5$ :  
 assumes *proper-symbols*  $zs$   
 shows *traces*  $tmB5 (tps\text{-ones } zs \ m) (esB5 (length \ zs) \ m) (tpsB5 \ zs \ m)$   
 ⟨proof⟩

**definition**  $tpsB6 :: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{tape list where}$   
 $tpsB6 \ zs \ m \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq m \text{ then } \mathbf{0}$   
 $\quad \text{else if } x \leq m + 1 + m * \text{length } zs \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $1)]$

**definition**  $esB56 :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esB56 \ n \ m \equiv \text{map} (Pair \ 1) (\text{rev } [0..<m + 1]) \ @ \ [(1, 0), (1, 1)]$

**definition**  $esB6 :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esB6 \ n \ m \equiv esB5 \ n \ m \ @ \ esB56 \ n \ m$

**lemma**  $\text{length-esB6}$ :  $\text{length} (esB6 \ n \ m) \leq 11 + 21 * m * m + 5 * m * m * n$   
 ⟨proof⟩

**lemma**  $tmB6$ :  
 assumes *proper-symbols*  $zs$   
 shows *traces*  $tmB6 (tps\text{-ones } zs \ m) (esB6 (length \ zs) \ m) (tpsB6 \ zs \ m)$   
 ⟨proof⟩

**definition**  $tpsB7 :: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{tape list where}$   
 $tpsB7 \ zs \ m \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq m + 1 + m * \text{length } zs \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $m + 1)]$

**definition**  $esB67 :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esB67 \ n \ m \equiv \text{map} (\lambda i. (1, 1 + Suc \ i)) [0..<m] \ @ \ [(1, 1 + m)]$

**definition**  $esB7 :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list where}$   
 $esB7 \ n \ m \equiv esB6 \ n \ m \ @ \ esB67 \ n \ m$

**lemma**  $\text{length-esB7}$ :  $\text{length} (esB7 \ n \ m) \leq 12 + 22 * m * m + 5 * m * m * n$   
 ⟨proof⟩

**lemma**  $tmB7$ :  
 assumes *proper-symbols*  $zs$   
 shows *traces*  $tmB7 (tps\text{-ones } zs \ m) (esB7 (length \ zs) \ m) (tpsB7 \ zs \ m)$   
 ⟨proof⟩

**definition**  $tpsB8 :: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{tape list where}$   
 $tpsB8 \ zs \ m \equiv$   
 $[(\lfloor zs \rfloor, 1),$

( $\lambda x$ . if  $x = 0$  then  $\triangleright$   
 else if  $x \leq m + 1 + m * \text{length } zs$  then  $\mathbf{1}$   
 else  $0$ ,  
 $1$ )]

**definition**  $esB78 :: nat \Rightarrow nat \Rightarrow (nat \times nat)$  list **where**  
 $esB78 \ n \ m \equiv \text{map } (Pair \ 1) \ (\text{rev } [0..<m + 1]) \ @ \ [(1, 0), (1, 1)]$

**definition**  $esB8 :: nat \Rightarrow nat \Rightarrow (nat \times nat)$  list **where**  
 $esB8 \ n \ m \equiv esB7 \ n \ m \ @ \ esB78 \ n \ m$

**lemma**  $\text{length-esB8}$ :  $\text{length } (esB8 \ n \ m) \leq 15 + 23 * m * m + 5 * m * m * n$   
 ⟨proof⟩

**lemma**  $tmB8$ :  
**assumes** *proper-symbols*  $zs$   
**shows**  $\text{traces } tmB8 \ (tps\text{-ones } zs \ m) \ (esB8 \ (\text{length } zs) \ m) \ (tpsB8 \ zs \ m)$   
 ⟨proof⟩

**lemma**  $tps\text{-ones-eq-tpsB8}$ :  $tpsB8 \ zs \ m = tps\text{-ones } zs \ (1 + m * (\text{length } zs + 1))$   
 ⟨proof⟩

**lemma**  $tmB$ :  
**assumes** *proper-symbols*  $zs$   
**shows**  $\text{traces } tmB \ (tps\text{-ones } zs \ m) \ (esB8 \ (\text{length } zs) \ m) \ (tps\text{-ones } zs \ (1 + m * (\text{length } zs + 1)))$   
 ⟨proof⟩

### 5.2.3 Appending a fixed number of symbols

The next Turing machine appends a constant number  $c$  of  $\mathbf{1}$  symbols to the output tape.

**definition**  $tmC :: nat \Rightarrow machine$  **where**  
 $tmC \ c \equiv$   
 $tm\text{-right-until } 1 \ \{\square\} \ ;;$   
 $tm\text{-write-repeat } 1 \ \mathbf{1} \ c \ ;;$   
 $tm\text{-cr } 1$

**lemma**  $tmC\text{-tm}$ :  $turing\text{-machine } 2 \ G \ (tmC \ c)$   
 ⟨proof⟩

**definition**  $tmC1 \equiv tm\text{-right-until } 1 \ \{\square\}$   
**definition**  $tmC2 \ c \equiv tmC1 \ ;; tm\text{-write-repeat } 1 \ \mathbf{1} \ c$   
**definition**  $tmC3 \ c \equiv tmC2 \ c \ ;; tm\text{-cr } 1$

**definition**  $tpsC1 :: symbol \ list \Rightarrow nat \Rightarrow tape \ list$  **where**  
 $tpsC1 \ zs \ m \equiv$   
 $[(zs, 1),$   
 $(\lambda x$ . if  $x = 0$  then  $\triangleright$   
 else if  $x \leq m$  then  $\mathbf{1}$   
 else  $0$ ,  
 $m + 1$ )]

**definition**  $esC1 :: nat \Rightarrow nat \Rightarrow (nat \times nat)$  list **where**  
 $esC1 \ n \ m \equiv \text{map } (\lambda i. (1, 1 + Suc \ i)) \ [0..<m] \ @ \ [(1, 1 + m)]$

**lemma**  $\text{length-esC1}$ :  $\text{length } (esC1 \ n \ m) = m + 1$   
 ⟨proof⟩

**lemma**  $tmC1$ :  
**assumes** *proper-symbols*  $zs$   
**shows**  $\text{traces } tmC1 \ (tps5 \ zs \ m) \ (esC1 \ (\text{length } zs) \ m) \ (tpsC1 \ zs \ m)$   
 ⟨proof⟩

**definition**  $tpsC2 :: symbol \ list \Rightarrow nat \Rightarrow nat \Rightarrow tape \ list$  **where**

$tpsC2\ zs\ m\ c \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq m + c \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $m + 1 + c)]$

**definition**  $esC12 :: nat \Rightarrow nat \Rightarrow nat \Rightarrow (nat \times nat)$  list **where**  
 $esC12\ n\ m\ c \equiv \text{map } (\lambda i. (1, m + 1 + \text{Suc } i)) [0..<c]$

**definition**  $esC2 :: nat \Rightarrow nat \Rightarrow nat \Rightarrow (nat \times nat)$  list **where**  
 $esC2\ n\ m\ c \equiv esC1\ n\ m\ @\ esC12\ n\ m\ c$

**lemma**  $\text{length-esC2}: \text{length } (esC2\ n\ m\ c) = m + 1 + c$   
 $\langle \text{proof} \rangle$

**lemma**  $tmC2:$   
**assumes** *proper-symbols*  $zs$   
**shows**  $\text{traces } (tmC2\ c)\ (tps5\ zs\ m)\ (esC2\ (\text{length } zs)\ m\ c)\ (tpsC2\ zs\ m\ c)$   
 $\langle \text{proof} \rangle$

**definition**  $tpsC3 :: \text{symbol list} \Rightarrow nat \Rightarrow nat \Rightarrow \text{tape list}$  **where**  
 $tpsC3\ zs\ m\ c \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lambda x. \text{if } x = 0 \text{ then } \triangleright$   
 $\quad \text{else if } x \leq m + c \text{ then } \mathbf{1}$   
 $\quad \text{else } 0,$   
 $1)]$

**definition**  $esC23 :: nat \Rightarrow nat \Rightarrow nat \Rightarrow (nat \times nat)$  list **where**  
 $esC23\ n\ m\ c \equiv \text{map } (\text{Pair } 1)\ (\text{rev } [0..<m + 1 + c])\ @\ [(1, 0), (1, 1)]$

**definition**  $esC3 :: nat \Rightarrow nat \Rightarrow nat \Rightarrow (nat \times nat)$  list **where**  
 $esC3\ n\ m\ c \equiv esC2\ n\ m\ c\ @\ esC23\ n\ m\ c$

**lemma**  $\text{length-esC3}: \text{length } (esC3\ n\ m\ c) = 2 * m + 2 * c + 4$   
 $\langle \text{proof} \rangle$

**lemma**  $tmC3:$   
**assumes** *proper-symbols*  $zs$   
**shows**  $\text{traces } (tmC3\ c)\ (tps5\ zs\ m)\ (esC3\ (\text{length } zs)\ m\ c)\ (tpsC3\ zs\ m\ c)$   
 $\langle \text{proof} \rangle$

**lemma**  $tpsC3\text{-eq-tps5}: tpsC3\ zs\ m\ c = tps5\ zs\ (m + c)$   
 $\langle \text{proof} \rangle$

**lemma**  $tmC3\text{-eq-}tmC: tmC3 = tmC$   
 $\langle \text{proof} \rangle$

**lemma**  $tmC:$   
**assumes** *proper-symbols*  $zs$   
**shows**  $\text{traces } (tmC\ c)\ (tps\text{-ones } zs\ m)\ (esC3\ (\text{length } zs)\ m\ c)\ (tps\text{-ones } zs\ (m + c))$   
 $\langle \text{proof} \rangle$

## 5.2.4 Polynomials of higher degree

In order to construct polynomials of arbitrary degree, we repeat the TM  $tmB$ .

**fun**  $tm\text{-degree} :: nat \Rightarrow \text{machine}$  **where**  
 $tm\text{-degree } 0 = []\ |$   
 $tm\text{-degree } (\text{Suc } d) = tm\text{-degree } d\ ;;\ tmB$

**lemma**  $tm\text{-degree-}tm: \text{turing-machine } 2\ G\ (tm\text{-degree } d)$   
 $\langle \text{proof} \rangle$

The number of **1** symbols the TM *tm-degree d* outputs on an input of length *n*:

**fun** *m-degree* :: *nat* ⇒ *nat* ⇒ *nat* **where**  
*m-degree n 0* = *n* |  
*m-degree n (Suc d)* = *1* + *m-degree n d* \* (*n* + *1*)

**fun** *es-degree* :: *nat* ⇒ *nat* ⇒ (*nat* \* *nat*) *list* **where**  
*es-degree n 0* = [] |  
*es-degree n (Suc d)* = *es-degree n d* @ *esB8 n (m-degree n d)*

**lemma** *tm-degree*:  
**assumes** *proper-symbols zs*  
**shows** *traces*  
(*tm-degree d*)  
(*tps-ones zs (length zs)*)  
(*es-degree (length zs) d*)  
(*tps-ones zs (m-degree (length zs) d)*)  
⟨*proof*⟩

A lower bound for the number of **1** symbols the TM *tm-degree d* outputs:

**lemma** *m-degree-ge-pow*: *m-degree n d* ≥ *n* ^ (*Suc d*)  
⟨*proof*⟩

An upper bound for the number of **1** symbols the TM *tm-degree d* outputs:

**lemma** *m-degree-poly*: *big-oh-poly* ( $\lambda n.$  *m-degree n d*)  
⟨*proof*⟩

**corollary** *m-degree-plus-const-poly*: *big-oh-poly* ( $\lambda n.$  *m-degree n d* + *c*)  
⟨*proof*⟩

**lemma** *length-es-degree*: *big-oh-poly* ( $\lambda n.$  *length (es-degree n d)*)  
⟨*proof*⟩

Putting together the TM *tmA*, the TM *tm-degree d* for some *d*, and the TM *tmC c* for some *c*, we get a family of TMs parameterized by *d* and *c*. These TMs construct all the polynomials we need.

**definition** *tm-poly* :: *nat* ⇒ *nat* ⇒ *machine* **where**  
*tm-poly d c* ≡ *tmA* ;; (*tm-degree d* ;; *tmC c*)

**lemma** *tm-poly-tm*: *turing-machine 2 G (tm-poly d c)*  
⟨*proof*⟩

**definition** *es-poly* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat* × *nat*) *list* **where**  
*es-poly n d c* ≡ *es5 n* @ *es-degree n d* @ *esC3 n (m-degree n d) c*

On an input of length *n* the Turing machine *tm-poly d c* outputs *m-degree n d* + *c* symbols **1**.

**lemma** *tm-poly*:  
**assumes** *proper-symbols zs*  
**shows** *traces*  
(*tm-poly d c*)  
(*tps0 zs*)  
(*es-poly (length zs) d c*)  
(*tps-ones zs (m-degree (length zs) d + c)*)  
⟨*proof*⟩

The Turing machines run in polynomial time because their traces have polynomial length:

**lemma** *length-es-poly*: *big-oh-poly* ( $\lambda n.$  *length (es-poly n d c)*)  
⟨*proof*⟩

The Turing machine *tm-poly d c* outputs *m-degree n c* + *c* many **1** symbols on an input of length *n*. Hence for every polynomially bounded function *f* there is such a Turing machine outputting at least *f(n)* symbols **1**.

**lemma** *m-degree-plus-const*:

**assumes** *big-oh-poly*  $f$   
**obtains**  $d\ c$  **where**  $\forall n. f\ n \leq m\text{-degree}\ n\ d + c$   
 $\langle$ *proof* $\rangle$

The Turing machines are oblivious.

**lemma** *tm-poly-oblivious*: *oblivious* (*tm-poly*  $d\ c$ )  
 $\langle$ *proof* $\rangle$

**end**

**definition** *start-tapes-2* :: *symbol list*  $\Rightarrow$  *tape list* **where**  
*start-tapes-2*  $zs \equiv$   
 $[[[zs], 0],$   
 $(\lambda i. \text{if } i = 0 \text{ then } \triangleright \text{ else } \square, 0)]$

**definition** *one-tapes-2* :: *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *tape list* **where**  
*one-tapes-2*  $zs\ m \equiv$   
 $[[[zs], 1],$   
 $([replicate\ m\ \mathbf{1}], 1)]$

The main result of this chapter. For every polynomially bounded function  $g$  there is a polynomial-time two-tape oblivious Turing machine that outputs at least  $g(n)$  symbols  $\mathbf{1}$  for every input length  $n$ .

**lemma** *polystructor*:  
**assumes** *big-oh-poly*  $g$  **and**  $G \geq 5$   
**shows**  $\exists M\ es\ f.$   
*turing-machine*  $2\ G\ M \wedge$   
*big-oh-poly*  $(\lambda n. \text{length}\ (es\ n)) \wedge$   
*big-oh-poly*  $f \wedge$   
 $(\forall n. g\ n \leq f\ n) \wedge$   
 $(\forall zs. \text{proper-symbols}\ zs \longrightarrow \text{traces}\ M\ (\text{start-tapes-2}\ zs)\ (es\ (\text{length}\ zs))\ (\text{one-tapes-2}\ zs\ (f\ (\text{length}\ zs))))$   
 $\langle$ *proof* $\rangle$

**end**

### 5.3 Existence of two-tape oblivious Turing machines

**theory** *Oblivious-2-Tape*  
**imports** *Oblivious-Polynomial NP*  
**begin**

In this section we show that for every polynomial-time multi-tape Turing machine there is a polynomial-time two-tape oblivious Turing machine that computes the same function and halts with its output tape head in cell number 1.

Consider a  $k$ -tape Turing machine  $M$  with polynomially bounded running time  $T$ . We construct a two-tape oblivious Turing machine  $S$  simulating  $M$  as follows.

Lemma *polystructor* from the previous section provides us with a polynomial-time two-tape oblivious TM and a function  $f(n) \geq T(n)$  such that the TM outputs  $\mathbf{1}^{f(n)}$  for all inputs of length  $n$ .

Executing this TM is the first thing our simulator does. The  $f(n)$  symbols  $\mathbf{1}$  mark the space  $S$  is going to use. Every cell  $i = 0, \dots, f(n) - 1$  of this space is to store a symbol that encodes a  $(2k + 2)$ -tuple consisting of:

- $k$  symbols from  $M$ 's alphabet representing the contents of all the  $i$ -th cells on the  $k$  tapes of  $M$ ;
- $k$  flags (called "head flags") signalling which of the  $k$  tape heads of  $M$  is in cell  $i$ ;
- a flag (called "counter flag") initialized with 0;
- a flag (called "start flag") signalling whether  $i = 0$ .

Together the counter flags are a unary counter from 0 to  $f(n)$ . They are toggled from left to right. The start flags never change. The symbols and the head flags represent the  $k$  tapes of  $M$  at some step of the

execution. By choice of  $f$  the TM  $M$  cannot use more than  $f(n)$  cells on any tape. So the space marked with  $\mathbf{1}$  symbols on the simulator's output tape suffices.

Next the simulator initializes the space of  $\mathbf{1}$  symbols with code symbols representing the start configuration of  $M$  for the input given to the simulator.

Then the main loop of the simulation performs  $f(n)$  iterations. In each iteration  $S$  performs one step of  $M$ 's computation. In order to do this it performs several left-to-right and right-to-left sweeps over all the  $f(n)$  cells in the formatted section of the output tape. A sweep will move the output tape head one cell right (respectively left) in each step. In this way the simulator's head positions at any time will only depend on  $f(n)$ , and hence on  $n$ . Thus the machine will be oblivious. Moreover  $f(n) \geq T(n)$ , and so  $M$  will be in the halting state after  $f(n)$  iterations of the simulation. Counting the iterations to  $f(n)$  is achieved via the counter flags.

Finally the simulator extracts from each code symbol the symbol corresponding to  $M$ 's output tape, thus reconstructing the output of  $M$  on the simulator's output tape. Thanks to the start flags, the simulator can easily locate the leftmost cell and put the output tape head one to the right of it, as required.

The construction heavily uses the memorization-in-states technique (see Chapter 2.5). At first the machine features  $2k + 1$  memorization tapes in addition to the input tape and output tape. The purpose of those tapes is to store  $M$ 's state and the symbols under the tape heads of  $M$  in the currently simulated step of  $M$ 's execution, as well as the  $k$  symbols to be written and head movements to be executed by the simulator.

The next predicate expresses that a Turing machine halts within a time bound depending on the length of the input. We did not have a need for this fairly basic predicate yet, because so far we were always interested in the halting configuration, too, and so the predicate *computes-in-time* sufficed.

**definition** *time-bound* :: machine  $\Rightarrow$  nat  $\Rightarrow$  (nat  $\Rightarrow$  nat)  $\Rightarrow$  bool **where**  
*time-bound*  $M$   $k$   $T$   $\equiv$   
 $\forall$   $zs$ . bit-symbols  $zs \longrightarrow$  fst (execute  $M$  (start-config  $k$   $zs$ ) ( $T$  (length  $zs$ ))) = length  $M$

**lemma** *time-bound-ge*:  
**assumes** *time-bound*  $M$   $k$   $T$  **and**  $\forall n$ . fnt  $n \geq T$   $n$   
**shows** *time-bound*  $M$   $k$  fnt  
 <proof>

The time bound also bounds the position of all the tape heads.

**lemma** *head-pos-le-time-bound*:  
**assumes** *turing-machine*  $k$   $G$   $M$   
**and** *time-bound*  $M$   $k$   $T$   
**and** bit-symbols  $zs$   
**and**  $j < k$   
**shows** execute  $M$  (start-config  $k$   $zs$ )  $t <\#\>$   $j \leq T$  (length  $zs$ )  
 <proof>

The entire construction will take place in a locale that assumes a polynomial-time  $k$ -tape Turing machine  $M$ .

**locale** *two-tape* =  
**fixes**  $M$  :: machine **and**  $k$   $G$  :: nat **and**  $T$  :: nat  $\Rightarrow$  nat  
**assumes** *tm-M*: *turing-machine*  $k$   $G$   $M$   
**and** *poly-T*: *big-oh-poly*  $T$   
**and** *time-bound-T*: *time-bound*  $M$   $k$   $T$   
**begin**

**lemma** *k-ge-2*:  $k \geq 2$   
 <proof>

**lemma** *G-ge-4*:  $G \geq 4$   
 <proof>

The construction of the simulator relies on the formatted space on the output tape to be large enough to hold the input. The size of the formatted space depends on the time bound  $T$ , which might be less than the length of the input. To ensure that the formatted space is large enough we increase the time bound while keeping it polynomial. The new bound is  $T'$ :

**definition**  $T' :: nat \Rightarrow nat$  **where**  
 $T' n \equiv T n + n$

**lemma** *poly-T'*: *big-oh-poly*  $T'$   
 $\langle proof \rangle$

**lemma** *time-bound-T'*: *time-bound*  $M k T'$   
 $\langle proof \rangle$

### 5.3.1 Encoding multiple tapes into one

The symbols on the output tape of the simulator are supposed to encode a  $(2k + 2)$ -tuple, where the first  $k$  elements assume one of the values in  $\{0, \dots, G - 1\}$ , whereas the other  $k + 2$  are flags with two possible values only. For uniformity we define an encoding where all elements range over  $G$  values and that works for tuples of every length.

**fun** *encode* ::  $nat list \Rightarrow nat$  **where**  
 $encode [] = 0$  |  
 $encode (x \# xs) = x + G * encode xs$

For every  $m \in \mathbb{N}$ , the encoding is a bijective mapping from  $\{0, \dots, G - 1\}^m$  to  $\{0, \dots, G^m - 1\}$ .

**lemma** *encode-surj*:  
**assumes**  $n < G \wedge m$   
**shows**  $\exists xs. length\ xs = m \wedge (\forall x \in set\ xs. x < G) \wedge encode\ xs = n$   
 $\langle proof \rangle$

**lemma** *encode-inj*:  
**assumes**  $\forall x \in set\ xs. x < G$   
**and**  $length\ xs = m$   
**and**  $\forall y \in set\ ys. y < G$   
**and**  $length\ ys = m$   
**and**  $encode\ xs = encode\ ys$   
**shows**  $xs = ys$   
 $\langle proof \rangle$

**lemma** *encode-less*:  
**assumes**  $\forall x \in set\ xs. x < G$   
**shows**  $encode\ xs < G \wedge (length\ xs)$   
 $\langle proof \rangle$

Decoding a number into an  $m$ -tuple of numbers is then a well-defined operation.

**definition** *decode* ::  $nat \Rightarrow nat \Rightarrow nat list$  **where**  
 $decode\ m\ n \equiv THE\ xs. encode\ xs = n \wedge length\ xs = m \wedge (\forall x \in set\ xs. x < G)$

**lemma** *decode*:  
**assumes**  $n < G \wedge m$   
**shows** *encode-decode*:  $encode\ (decode\ m\ n) = n$   
**and** *length-decode*:  $length\ (decode\ m\ n) = m$   
**and** *decode-less-G*:  $\forall x \in set\ (decode\ m\ n). x < G$   
 $\langle proof \rangle$

**lemma** *decode-encode*:  $\forall x \in set\ xs. x < G \implies decode\ (length\ xs)\ (encode\ xs) = xs$   
 $\langle proof \rangle$

The simulator will access and update components of the encoded symbol.

**definition** *encode-nth* ::  $nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$  **where**  
 $encode-nth\ m\ n\ j \equiv decode\ m\ n ! j$

**definition** *encode-upd* ::  $nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$  **where**  
 $encode-upd\ m\ n\ j\ x \equiv encode\ ((decode\ m\ n) [j:=x])$

**lemma** *encode-nth-less*:  
**assumes**  $n < G \wedge m$  **and**  $j < m$

**shows**  $encode\text{-}nth\ m\ n\ j < G$   
 ⟨proof⟩

For the symbols the simulator actually uses, we fix  $m = 2k+2$  and reserve the symbols  $\triangleright$  and  $\square$ , effectively shifting the symbols by two. We call the symbols  $\{2, \dots, G^{2k+2} + 2\}$  “code symbols”.

**definition**  $enc :: symbol\ list \Rightarrow symbol\ where$   
 $enc\ xs \equiv encode\ xs + 2$

**definition**  $dec :: symbol \Rightarrow symbol\ list\ where$   
 $dec\ n \equiv decode\ (2 * k + 2)\ (n - 2)$

**lemma**  $dec$ :  
**assumes**  $n > 1$  **and**  $n < G \wedge (2 * k + 2) + 2$   
**shows**  $enc\text{-}dec$ :  $enc\ (dec\ n) = n$   
**and**  $length\text{-}dec$ :  $length\ (dec\ n) = 2 * k + 2$   
**and**  $dec\text{-}less\text{-}G$ :  $\forall x \in set\ (dec\ n). x < G$   
 ⟨proof⟩

**lemma**  $dec\text{-}enc$ :  $\forall x \in set\ xs. x < G \Longrightarrow length\ xs = 2 * k + 2 \Longrightarrow dec\ (enc\ xs) = xs$   
 ⟨proof⟩

**definition**  $enc\text{-}nth :: nat \Rightarrow nat \Rightarrow nat\ where$   
 $enc\text{-}nth\ n\ j \equiv dec\ n ! j$

**lemma**  $enc\text{-}nth$ :  $\forall x \in set\ xs. x < G \Longrightarrow length\ xs = 2 * k + 2 \Longrightarrow enc\text{-}nth\ (enc\ xs)\ j = xs ! j$   
 ⟨proof⟩

**lemma**  $enc\text{-}nth\text{-}dec$ :  
**assumes**  $n > 1$  **and**  $n < G \wedge (2 * k + 2) + 2$   
**shows**  $enc\text{-}nth\ n\ j = (dec\ n) ! j$   
 ⟨proof⟩

**abbreviation**  $is\text{-}code :: nat \Rightarrow bool\ where$   
 $is\text{-}code\ n \equiv n < G \wedge (2 * k + 2) + 2 \wedge 1 < n$

**lemma**  $enc\text{-}nth\text{-}less$ :  
**assumes**  $is\text{-}code\ n$  **and**  $j < 2 * k + 2$   
**shows**  $enc\text{-}nth\ n\ j < G$   
 ⟨proof⟩

**lemma**  $enc\text{-}less$ :  $\forall x \in set\ xs. x < G \Longrightarrow length\ xs = 2 * k + 2 \Longrightarrow enc\ xs < G \wedge (2 * k + 2) + 2$   
 ⟨proof⟩

**definition**  $enc\text{-}upd :: nat \Rightarrow nat \Rightarrow nat \Rightarrow nat\ where$   
 $enc\text{-}upd\ n\ j\ x \equiv enc\ ((dec\ n)\ [j := x])$

**lemma**  $enc\text{-}upd\text{-}is\text{-}code$ :  
**assumes**  $is\text{-}code\ n$  **and**  $j < 2 * k + 2$  **and**  $x < G$   
**shows**  $is\text{-}code\ (enc\text{-}upd\ n\ j\ x)$   
 ⟨proof⟩

The code symbols require the simulator to have an alphabet of at least size  $G^{2k+2} + 2$ . On top of that we want to store on a memorization tape the state that  $M$  is in. So the alphabet must have at least  $length\ M + 1$  symbols. Both conditions are met by the simulator having an alphabet of size  $G'$ :

**definition**  $G' :: nat\ where$   
 $G' \equiv G \wedge (2 * k + 2) + 2 + length\ M$

**lemma**  $G'\text{-}ge\text{-}6$ :  $G' \geq 6$   
 ⟨proof⟩

**corollary**  $G'\text{-}ge$ :  $G' \geq 4\ G' \geq 5$   
 ⟨proof⟩

**lemma**  $G'$ -ge- $G$ :  $G' \geq G$   
 ⟨proof⟩

**corollary**  $enc$ -less- $G'$ :  $\forall x \in set\ xs. x < G \implies length\ xs = 2 * k + 2 \implies enc\ xs < G'$   
 ⟨proof⟩

**lemma**  $enc$ -greater:  $enc\ xs > 1$   
 ⟨proof⟩

### 5.3.2 Construction of the simulator Turing machine

The simulator is a sequence of three Turing machines: The “formatter”, which initializes the output tape; the loop, which simulates the TM  $M$  for polynomially many steps; and a cleanup TM, which makes the output tape look like the output tape of  $M$ . All these machines are discussed in order in the following subsections.

The simulator will start with  $2k + 1$  memorization tapes for a total of  $2k + 3$  tapes. The simulation action will take place on the output tape.

#### Initializing the simulator’s tapes

The function  $T'$  is polynomially bounded and therefore there is a polynomial-time two-tape oblivious Turing machine that outputs at least  $T'(n)$  symbols **1** on an input of length  $n$ , as we have proven in the previous section (lemma *polystructor*). We now obtain such a Turing machine together with its running time bound and trace. This TM will be one of our blocks for building the simulator TM. We will call it the “formatter”.

**definition**  $fnt$ -es- $pM$  ::  $(nat \Rightarrow nat) \times (nat \Rightarrow (nat \times nat)\ list) \times machine$  **where**

$fnt$ -es- $pM \equiv SOME\ tec.$   
 $turing$ -machine 2  $G'$  (snd (snd tec))  $\wedge$   
 $big$ -oh-poly  $(\lambda n. length\ ((fst\ (snd\ tec))\ n)) \wedge$   
 $big$ -oh-poly (fst tec)  $\wedge$   
 $(\forall n. T'\ n \leq (fst\ tec)\ n) \wedge$   
 $(\forall zs. proper$ -symbols  $zs \longrightarrow traces\ (snd\ (snd\ tec))\ (start$ -tapes-2  $zs)\ ((fst\ (snd\ tec))\ (length\ zs))\ (one$ -tapes-2  $zs\ ((fst\ tec)\ (length\ zs))))$

**lemma**  $polystructor'$ :

**fixes**  $GG$  ::  $nat$  **and**  $g$  ::  $nat \Rightarrow nat$   
**assumes**  $big$ -oh-poly  $g$  **and**  $GG \geq 5$   
**shows**  $\exists f$ -es- $M.$   
 $turing$ -machine 2  $GG$  (snd (snd  $f$ -es- $M$ ))  $\wedge$   
 $big$ -oh-poly  $(\lambda n. length\ ((fst\ (snd\ f$ -es- $M))\ n)) \wedge$   
 $big$ -oh-poly (fst  $f$ -es- $M$ )  $\wedge$   
 $(\forall n. g\ n \leq (fst\ f$ -es- $M)\ n) \wedge$   
 $(\forall zs. proper$ -symbols  $zs \longrightarrow traces\ (snd\ (snd\ f$ -es- $M))\ (start$ -tapes-2  $zs)\ ((fst\ (snd\ f$ -es- $M))\ (length\ zs))$   
 $(one$ -tapes-2  $zs\ ((fst\ f$ -es- $M)\ (length\ zs))))$   
 ⟨proof⟩

**lemma**  $fnt$ -es- $pM$ :  $turing$ -machine 2  $G'$  (snd (snd  $fnt$ -es- $pM$ ))  $\wedge$

$big$ -oh-poly  $(\lambda n. length\ ((fst\ (snd\ fnt$ -es- $pM))\ n)) \wedge$   
 $big$ -oh-poly (fst  $fnt$ -es- $pM$ )  $\wedge$   
 $(\forall n. T'\ n \leq (fst\ fnt$ -es- $pM)\ n) \wedge$   
 $(\forall zs. proper$ -symbols  $zs \longrightarrow traces\ (snd\ (snd\ fnt$ -es- $pM))\ (start$ -tapes-2  $zs)\ ((fst\ (snd\ fnt$ -es- $pM))\ (length\ zs))$   
 $(one$ -tapes-2  $zs\ ((fst\ fnt$ -es- $pM)\ (length\ zs))))$   
 ⟨proof⟩

**definition**  $fnt$  ::  $nat \Rightarrow nat$  **where**

$fnt \equiv fst\ fnt$ -es- $pM$

**definition**  $es$ - $fnt$  ::  $nat \Rightarrow (nat \times nat)\ list$  **where**

$es$ - $fnt \equiv fst\ (snd\ fnt$ -es- $pM)$

**definition**  $tm\text{-}fmt :: machine$  **where**  
 $tm\text{-}fmt \equiv snd (snd\ fmt\text{-}es\text{-}pM)$

The formatter TM is  $tm\text{-}fmt$ , the number of **1** symbols written to the output tape on input of length  $n$  is  $fmt\ n$ , and the trace on inputs of length  $n$  is  $es\text{-}fmt\ n$ .

**lemma**  $fmt$ :

$turing\text{-}machine\ 2\ G'\ tm\text{-}fmt$   
 $big\text{-}oh\text{-}poly\ (\lambda n. length\ (es\text{-}fmt\ n))$   
 $big\text{-}oh\text{-}poly\ fmt$   
 $\bigwedge n. T^1\ n \leq\ fmt\ n$   
 $\bigwedge zs. proper\text{-}symbols\ zs \implies$   
 $traces\ tm\text{-}fmt\ (start\text{-}tapes\text{-}2\ zs)\ (es\text{-}fmt\ (length\ zs))\ (one\text{-}tapes\text{-}2\ zs\ (fmt\ (length\ zs)))$   
 $\langle proof \rangle$

**lemma**  $fmt\text{-}ge\text{-}n$ :  $fmt\ n \geq n$   
 $\langle proof \rangle$

The formatter is a two-tape TM. The first incarnation of the simulator will have two tapes and  $2k + 1$  memorization tapes. So for now we formally need to extend the formatter to  $2k + 3$  tapes:

**definition**  $tm1 \equiv append\text{-}tapes\ 2\ (2 * k + 3)\ tm\text{-}fmt$

**lemma**  $tm1\text{-}tm$ :  $turing\text{-}machine\ (2 * k + 3)\ G'\ tm1$   
 $\langle proof \rangle$

Next we replace all non-blank symbols on the output tape by code symbols representing the tuple of  $2k + 2$  zeros.

**definition**  $tm1\text{-}2 \equiv tm\text{-}const\text{-}until\ 1\ 1\ \{\square\}\ (enc\ (replicate\ k\ 0\ @\ replicate\ k\ 0\ @\ [0, 0]))$

**lemma**  $tm1\text{-}2\text{-}tm$ :  $turing\text{-}machine\ (2 * k + 3)\ G'\ tm1\text{-}2$   
 $\langle proof \rangle$

**definition**  $tm2 \equiv tm1\ ;\ ;\ tm1\text{-}2$

**lemma**  $tm2\text{-}tm$ :  $turing\text{-}machine\ (2 * k + 3)\ G'\ tm2$   
 $\langle proof \rangle$

**definition**  $tm3 \equiv tm2\ ;\ ;\ tm\text{-}start\ 1$

**lemma**  $tm3\text{-}tm$ :  $turing\text{-}machine\ (2 * k + 3)\ G'\ tm3$   
 $\langle proof \rangle$

Back at the start symbol of the output tape, we replace it by the code symbol for the tuple  $1^k 1^k 0 1$ . The first  $k$  ones represent that initially the  $k$  tapes of  $M$  have the start symbol (numerically 1) in the leftmost cell. The second run of  $k$  ones represent that initially all  $k$  tapes have their tape heads in the leftmost cell. The following 0 is the first bit of the unary counter, currently set to zero. The final flag 1 signals that this is the leftmost cell. Unlike the start symbols this signal flag cannot be overwritten by  $M$ .

**definition**  $tm4 \equiv tm3\ ;\ ;\ tm\text{-}write\ 1\ (enc\ (replicate\ k\ 1\ @\ replicate\ k\ 1\ @\ [0, 1]))$

**lemma**  $tm3\text{-}4\text{-}tm$ :  $turing\text{-}machine\ (2 * k + 3)\ G'\ (tm\text{-}write\ 1\ (enc\ (replicate\ k\ 1\ @\ replicate\ k\ 1\ @\ [0, 1])))$   
 $\langle proof \rangle$

**lemma**  $tm4\text{-}tm$ :  $turing\text{-}machine\ (2 * k + 3)\ G'\ tm4$   
 $\langle proof \rangle$

**definition**  $tm5 \equiv tm4\ ;\ ;\ tm\text{-}right\ 1$

**lemma**  $tm5\text{-}tm$ :  $turing\text{-}machine\ (2 * k + 3)\ G'\ tm5$   
 $\langle proof \rangle$

So far the simulator's output tape encodes  $k$  tapes that are empty but for the start symbols. To represent the start configuration of  $M$ , we need to copy the contents of the input tape to the output tape. The

following TM moves the work head to the first blank while copying the input tape content. Here we exploit  $T'(n) \geq n$ , which implies that the formatted section is long enough to hold the input.

**definition**  $tm5-6 \equiv tm-trans-until\ 0\ 1\ \{0\}\ (\lambda h. enc\ (h\ mod\ G\ \# replicate\ (k - 1)\ 0\ @ replicate\ k\ 0\ @ [0, 0]))$

**definition**  $tm6 \equiv tm5\ ;;\ tm5-6$

**lemma**  $tm5-6-tm: turing-machine\ (2 * k + 3)\ G'\ tm5-6$   
 $\langle proof \rangle$

**lemma**  $tm6-tm: turing-machine\ (2 * k + 3)\ G'\ tm6$   
 $\langle proof \rangle$

Since we have overwritten the leftmost cell of the output tape with some code symbol, we cannot return to it using  $tm-start$ . But we can use  $tm-left-until$  with a special set of symbols:

**abbreviation**  $StartSym :: symbol\ set\ \mathbf{where}$   
 $StartSym \equiv \{y. y < G \wedge (2 * k + 2) + 2 \wedge y > 1 \wedge dec\ y\ !\ (2 * k + 1) = 1\}$

**abbreviation**  $tm-left-until1 \equiv tm-left-until\ StartSym\ 1$

**lemma**  $tm-left-until1-tm: turing-machine\ (2 * k + 3)\ G'\ tm-left-until1$   
 $\langle proof \rangle$

**definition**  $tm7 \equiv tm6\ ;;\ tm-left-until1$

**lemma**  $tm7-tm: turing-machine\ (2 * k + 3)\ G'\ tm7$   
 $\langle proof \rangle$

Tape number 2 is meant to memorize  $M$ 's state during the simulation. Initially the state is the start state, that is, zero.

**definition**  $tm8 \equiv tm7\ ;;\ tm-write\ 2\ 0$

**lemma**  $tm8-tm: turing-machine\ (2 * k + 3)\ G'\ tm8$   
 $\langle proof \rangle$

We also initialize memorization tapes  $3, \dots, 2k + 2$  to zero. This concludes the initialization of the simulator's tapes.

**definition**  $tm9 \equiv tm8\ ;;\ tm-write-many\ \{3..<2 * k + 3\}\ 0$

**lemma**  $tm9-tm: turing-machine\ (2 * k + 3)\ G'\ tm9$   
 $\langle proof \rangle$

## The loop

The core of the simulator is a loop whose body is executed  $fmt\ n$  many times. Each iteration simulates one step of the Turing machine  $M$ . For the analysis of the loop we describe the  $2k + 3$  tapes in the form  $[a, b, c] @ map\ f1\ [0..<k] @ map\ f2\ [0..<k]$ .

**lemma**  $threeplus2k-2:$   
**assumes**  $3 \leq j \wedge j < k + 3$   
**shows**  $([a, b, c] @ map\ f1\ [0..<k] @ map\ f2\ [0..<k]) ! j = f1\ (j - 3)$   
 $\langle proof \rangle$

**lemma**  $threeplus2k-3:$   
**assumes**  $k + 3 \leq j \wedge j < 2 * k + 3$   
**shows**  $([a, b, c] @ map\ f1\ [0..<k] @ map\ f2\ [0..<k]) ! j = f2\ (j - k - 3)$   
 $\langle proof \rangle$

To ensure the loop runs for  $fmt\ n$  iterations we increment the unary counter in the code symbols in each iteration. The loop terminates when there are no more code symbols with an unset counter flag. The TM that prepares the loop condition sweeps the output tape left-to-right searching for the first symbol that is either blank or has an unset counter flag. The condition then checks for which of the two cases occurred. This is the condition TM:

**definition**  $tmC \equiv tm\text{-right-until } 1 \{y. y < G \wedge (2 * k + 2) + 2 \wedge (y = 0 \vee dec\ y ! (2 * k) = 0)\}$

**lemma**  $tmC\text{-tm}$ : *turing-machine*  $(2 * k + 3) G' tmC$   
 ⟨proof⟩

At the start of the iteration, the memorization tape 2 has the state of  $M$ , and all other memorization tapes contain 0. The output tape head is at the leftmost code symbol with unset counter flag. The first order of business is to move back to the beginning of the output tape.

**definition**  $tmL1 \equiv tm\text{-left-until } 1$

**lemma**  $tmL1\text{-tm}$ : *turing-machine*  $(2 * k + 3) G' tmL1$   
 ⟨proof⟩

Then the output tape head sweeps right until it encounters a blank. During the sweep the Turing machine checks for any set head flags, and if it finds the one for tape  $j$  set, it memorizes the symbol for tape  $j$  on tape  $3 + k + j$ . The next command performs this operation:

**definition**  $cmdL2$  :: *command where*

$cmdL2\ rs \equiv$   
 (if  $rs ! 1 = \square$   
 then  $(1, zip\ rs\ (replicate\ (2*k+3)\ Stay))$   
 else  
 $(0,$   
 $[(rs!0, Stay), (rs!1, Right), (rs!2, Stay)] @$   
 $(map\ (\lambda j. (rs ! (j + 3), Stay)) [0..<k]) @$   
 $(map\ (\lambda j. (if\ 1 < rs ! 1 \wedge rs ! 1 < G \wedge (2*k+2)+2 \wedge enc\text{-nth}\ (rs!1)\ (k+j) = 1\ then\ enc\text{-nth}\ (rs!1)\ j\ else$   
 $rs!(3+k+j), Stay)) [0..<k]))$

**lemma**  $cmdL2\text{-at-}0$ :

**assumes**  $rs ! 1 = 0$  **and**  $j < 2 * k + 3$  **and**  $length\ rs = 2 * k + 3$

**shows**  $snd\ (cmdL2\ rs) ! j = (rs ! j, Stay)$

⟨proof⟩

**lemma**  $cmdL2\text{-at-}3$ :

**assumes**  $rs ! 1 \neq \square$  **and**  $3 \leq j \wedge j < k + 3$

**shows**  $cmdL2\ rs [!] j = (rs ! j, Stay)$

⟨proof⟩

**lemma**  $cmdL2\text{-at-}4$ :

**assumes**  $rs ! 1 \neq \square$  **and**  $k + 3 \leq j \wedge j < 2 * k + 3$

**shows**  $cmdL2\ rs [!] j =$

$(if\ 1 < rs ! 1 \wedge rs ! 1 < G \wedge (2*k+2)+2 \wedge enc\text{-nth}\ (rs ! 1)\ (j-3) = 1$

$then\ enc\text{-nth}\ (rs ! 1)\ (j-k-3)$

$else\ rs ! j, Stay)$

⟨proof⟩

**lemma**  $cmdL2\text{-at-}4''$ :

**assumes**  $rs ! 1 \neq \square$

**and**  $k + 3 \leq j \wedge j < 2 * k + 3$

**and**  $\neg (1 < rs ! 1 \wedge rs ! 1 < G \wedge (2*k+2)+2)$

**shows**  $cmdL2\ rs [!] j = (rs ! j, Stay)$

⟨proof⟩

**lemma**  $turing\text{-command-}cmdL2$ : *turing-command*  $(2 * k + 3) 1 G' cmdL2$

⟨proof⟩

**definition**  $tmL1\text{-}2 \equiv [cmdL2]$

**lemma**  $tmL1\text{-}2\text{-tm}$ : *turing-machine*  $(2 * k + 3) G' tmL1\text{-}2$

⟨proof⟩

**definition**  $tmL2 \equiv tmL1 ;; tmL1\text{-}2$

**lemma**  $tmL2\text{-tm}$ : *turing-machine*  $(2 * k + 3) G' tmL2$

*<proof>*

The memorization tapes  $3, \dots, 2 + k$  will store the head movements for tapes  $0, \dots, k - 1$  of  $M$ . The directions are encoded as symbols thus:

**definition** *direction-to-symbol* :: *direction*  $\Rightarrow$  *symbol* **where**  
*direction-to-symbol*  $m \equiv (\text{case } m \text{ of Left} \Rightarrow \square \mid \text{Stay} \Rightarrow \triangleright \mid \text{Right} \Rightarrow \mathbf{0})$

**lemma** *direction-to-symbol-less*: *direction-to-symbol*  $m < 3$   
*<proof>*

At this point in the iteration the memorization tapes  $k + 3, \dots, 2k + 2$  contain the symbols under the  $k$  tape heads of  $M$ , and tape 2 contains the state  $M$  is in. Therefore all information is available to determine the actions  $M$  is taking in the step currently simulated. The next command has the entire behavior of  $M$  “hard-coded” and causes the actions to be stored on memorization tapes  $3, \dots, 2k + 2$ : the output symbols on the tapes  $k + 3, \dots, 2k + 2$ , and the  $k$  head movements on the tapes  $3, \dots, k + 2$ . The follow-up state will again be memorized on tape 2. All this happens in one step of the simulator machine.

**definition** *cmdL3* :: *command* **where**

*cmdL3*  $rs \equiv$   
 $(1,$   
 $[(rs ! 0, \text{Stay}),$   
 $(rs ! 1, \text{Stay}),$   
 $(\text{if } rs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) rs). h < G)$   
 $\text{then } \text{fst } ((M ! (rs ! 2)) (\text{drop } (k + 3) rs))$   
 $\text{else } rs ! 2, \text{Stay})] @$   
 $\text{map}$   
 $(\lambda j. (\text{if } rs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) rs). h < G) \text{ then } \text{direction-to-symbol } ((M ! (rs ! 2))$   
 $(\text{drop } (k + 3) rs) [\sim] j) \text{ else } 1, \text{Stay}))$   
 $[0..<k] @$   
 $\text{map } (\lambda j. (\text{if } rs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) rs). h < G) \text{ then } ((M ! (rs ! 2)) (\text{drop } (k + 3) rs)$   
 $[\cdot] j) \text{ else } rs ! (k + 3 + j), \text{Stay})) [0..<k])$

**lemma** *cmdL3-at-2a*:

**assumes**  $gs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) gs). h < G)$   
**shows**  $\text{cmdL3 } gs [!] 2 = (\text{fst } ((M ! (gs ! 2)) (\text{drop } (k + 3) gs)), \text{Stay})$   
*<proof>*

**lemma** *cmdL3-at-2b*:

**assumes**  $\neg (gs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) gs). h < G))$   
**shows**  $\text{cmdL3 } gs [!] 2 = (gs ! 2, \text{Stay})$   
*<proof>*

**lemma** *cmdL3-at-3a*:

**assumes**  $3 \leq j \wedge j < k + 3$   
**and**  $gs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) gs). h < G)$   
**shows**  $\text{cmdL3 } gs [!] j = (\text{direction-to-symbol } ((M ! (gs ! 2)) (\text{drop } (k + 3) gs) [\sim] (j - 3)), \text{Stay})$   
*<proof>*

**lemma** *cmdL3-at-3b*:

**assumes**  $3 \leq j \wedge j < k + 3$   
**and**  $\neg (gs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) gs). h < G))$   
**shows**  $\text{cmdL3 } gs [!] j = (1, \text{Stay})$   
*<proof>*

**lemma** *cmdL3-at-4a*:

**assumes**  $k + 3 \leq j \wedge j < 2 * k + 3$   
**and**  $gs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) gs). h < G)$   
**shows**  $\text{cmdL3 } gs [!] j = ((M ! (gs ! 2)) (\text{drop } (k + 3) gs) [\cdot] (j - k - 3), \text{Stay})$   
*<proof>*

**lemma** *cmdL3-at-4b*:

**assumes**  $k + 3 \leq j \wedge j < 2 * k + 3$   
**and**  $\neg (gs ! 2 < \text{length } M \wedge (\forall h \in \text{set } (\text{drop } (k + 3) gs). h < G))$

**shows**  $cmdL3\ gs\ [!]\ j = (gs\ !\ j,\ Stay)$   
 ⟨proof⟩

**lemma**  $cmdL3\text{-if}\text{-comm}$ :

**assumes**  $length\ gs = 2 * k + 3$  **and**  $gs\ !\ 2 < length\ M \wedge (\forall h \in set\ (drop\ (k + 3)\ gs).\ h < G)$   
**shows**  $length\ ([!!]\ (M\ !\ (gs\ !\ 2))\ (drop\ (k + 3)\ gs)) = k$   
**and**  $\bigwedge j.\ j < k \implies (M\ !\ (gs\ !\ 2))\ (drop\ (k + 3)\ gs)\ [.] j < G$   
 ⟨proof⟩

**lemma**  $turing\text{-command}\text{-}cmdL3$ :  $turing\text{-command}\ (2 * k + 3)\ 1\ G'\ cmdL3$   
 ⟨proof⟩

**definition**  $tmL2\text{-}3 \equiv [cmdL3]$

**lemma**  $tmL2\text{-}3\text{-tm}$ :  $turing\text{-machine}\ (2 * k + 3)\ G'\ tmL2\text{-}3$   
 ⟨proof⟩

**definition**  $tmL3 \equiv tmL2\ ;\ ;\ tmL2\text{-}3$

**lemma**  $tmL3\text{-tm}$ :  $turing\text{-machine}\ (2 * k + 3)\ G'\ tmL3$   
 ⟨proof⟩

Next the output tape head sweeps left to the beginning of the tape, otherwise doing nothing.

**definition**  $tmL4 \equiv tmL3\ ;\ ;\ tm\text{-left}\text{-until}\ 1$

**lemma**  $tmL4\text{-tm}$ :  $turing\text{-machine}\ (2 * k + 3)\ G'\ tmL4$   
 ⟨proof⟩

The next four commands  $cmdL4$ ,  $cmdL5$ ,  $cmdL6$ ,  $cmdL7$  are parameterized by  $jj = 0, \dots, k - 1$ . Their job is applying the write operation and head movement for tape  $jj$  of  $M$ . The entire block of commands will then be executed  $k$  times, once for each  $jj$ .

The first of these commands sweeps right again and applies the write operation for tape  $jj$ , which is memorized on tape  $3 + k + jj$ . To this end it checks for head flags and updates the code symbol component  $jj$  with the contents of tape  $3 + k + jj$  when the head flag for tape  $jj$  is set.

**definition**  $cmdL5\ jj\ rs \equiv$

$if\ rs\ !\ 1 = \square$   
 then  $(1,\ zip\ rs\ (replicate\ (2 * k + 3)\ Stay))$   
 else  
 $(0,$   
 $[(rs\ !\ 0,\ Stay),$   
 $(if\ is\text{-code}\ (rs\ !\ 1) \wedge rs\ !\ (3 + k + jj) < G \wedge enc\text{-nth}\ (rs\ !\ 1)\ (k + jj) = 1$   
 then  $enc\text{-upd}\ (rs\ !\ 1)\ jj\ (rs\ !\ (3 + k + jj))$   
 else  $rs\ !\ 1,$   
 $Right),$   
 $(rs\ !\ 2,\ Stay)]\ @$   
 $(map\ (\lambda j.\ (rs\ !\ (j + 3),\ Stay))\ [0..<k])\ @$   
 $(map\ (\lambda j.\ (rs\ !\ (3 + k + j),\ Stay))\ [0..<k]))$

**lemma**  $cmdL5\text{-eq}\ 0$ :

**assumes**  $j < 2 * k + 3$  **and**  $length\ gs = 2 * k + 3$  **and**  $gs\ !\ 1 = 0$   
**shows**  $cmdL5\ jj\ gs\ [!]\ j = (gs\ !\ j,\ Stay)$   
 ⟨proof⟩

**lemma**  $cmdL5\text{-at}\ 0$ :

**assumes**  $gs\ !\ 1 \neq 0$   
**shows**  $cmdL5\ jj\ gs\ [!]\ 0 = (gs\ !\ 0,\ Stay)$   
 ⟨proof⟩

**lemma**  $cmdL5\text{-at}\ 1$ :

**assumes**  $gs\ !\ 1 \neq 0$   
**and**  $is\text{-code}\ (gs\ !\ 1) \wedge gs\ !\ (3 + k + jj) < G \wedge enc\text{-nth}\ (gs\ !\ 1)\ (k + jj) = 1$   
**shows**  $cmdL5\ jj\ gs\ [!]\ 1 = (enc\text{-upd}\ (gs\ !\ 1)\ jj\ (gs\ !\ (3 + k + jj)),\ Right)$   
 ⟨proof⟩

**lemma** *cmdL5-at-1-else*:

**assumes**  $gs ! 1 \neq 0$   
**and**  $\neg (is\_code (gs ! 1) \wedge gs ! (3+k+jj) < G \wedge enc\_nth (gs!1) (k+jj) = 1)$   
**shows**  $cmdL5\ jj\ gs\ [!]\ 1 = (gs ! 1, Right)$   
 $\langle proof \rangle$

**lemma** *cmdL5-at-2*:

**assumes**  $gs ! 1 \neq 0$   
**shows**  $cmdL5\ jj\ gs\ [!]\ 2 = (gs ! 2, Stay)$   
 $\langle proof \rangle$

**lemma** *cmdL5-at-3*:

**assumes**  $gs ! 1 \neq 0$  **and**  $3 \leq j \wedge j < 3 + k$   
**shows**  $cmdL5\ jj\ gs\ [!]\ j = (gs ! j, Stay)$   
 $\langle proof \rangle$

**lemma** *cmdL5-at-4*:

**assumes**  $gs ! 1 \neq 0$  **and**  $3 + k \leq j \wedge j < 2 * k + 3$   
**shows**  $cmdL5\ jj\ gs\ [!]\ j = (gs ! j, Stay)$   
 $\langle proof \rangle$

**lemma** *turing-command-cmdL5*:

**assumes**  $jj < k$   
**shows**  $turing\_command\ (2 * k + 3)\ 1\ G'\ (cmdL5\ jj)$   
 $\langle proof \rangle$

**definition** *tmL45* :: *nat*  $\Rightarrow$  *machine* **where**

$tmL45\ jj \equiv [cmdL5\ jj]$

**lemma** *tmL45-tm*:

**assumes**  $jj < k$   
**shows**  $turing\_machine\ (2 * k + 3)\ G'\ (tmL45\ jj)$   
 $\langle proof \rangle$

We move the output tape head one cell to the left.

**definition** *tmL46*  $jj \equiv tmL45\ jj\ ;;\ tm\_left\ 1$

**lemma** *tmL46-tm*:

**assumes**  $jj < k$   
**shows**  $turing\_machine\ (2 * k + 3)\ G'\ (tmL46\ jj)$   
 $\langle proof \rangle$

The next command sweeps left and applies the head movement for tape  $jj$  if this is a movement to the left. To this end it checks for a set head flag in component  $k + jj$  of the code symbol and clears it. It also memorizes that it just cleared the head flag by changing the symbol on memorization tape  $3 + jj$  to the number 3, which is not used to encode any actual head movement. In the next step of the sweep it will recognize this 3 and set the head flag in component  $k + jj$  of the code symbol. The net result is that the head flag for tape  $jj$  has moved one cell to the left.

**abbreviation** *is-beginning* :: *symbol*  $\Rightarrow$  *bool* **where**

$is\_beginning\ y \equiv is\_code\ y \wedge dec\ y ! (2 * k + 1) = 1$

**definition** *cmdL7* :: *nat*  $\Rightarrow$  *command* **where**

$cmdL7\ jj\ rs \equiv$   
 $(if\ is\_beginning\ (rs ! 1)\ then\ 1\ else\ 0,$   
 $if\ rs ! (3 + jj) = \square \wedge enc\_nth\ (rs ! 1)\ (k + jj) = 1 \wedge is\_code\ (rs ! 1) \wedge \neg is\_beginning\ (rs ! 1)\ then$   
 $[(rs ! 0, Stay),$   
 $(enc\_upd\ (rs ! 1)\ (k + jj)\ 0, Left),$   
 $(rs ! 2, Stay)]\ @$   
 $(map\ (\lambda j. (if\ j = jj\ then\ 3\ else\ rs ! (j + 3), Stay))\ [0..<k])\ @$   
 $(map\ (\lambda j. (rs ! (3 + k + j), Stay))\ [0..<k])$   
 $else\ if\ rs ! (3 + jj) = 3 \wedge is\_code\ (rs ! 1)\ then$   
 $[(rs ! 0, Stay),$

$(enc-upd (rs ! 1) (k + jj) 1, Left),$   
 $(rs ! 2, Stay)] @$   
 $(map (\lambda j. (if j = jj then 0 else rs ! (j + 3), Stay)) [0..<k]) @$   
 $(map (\lambda j. (rs ! (3 + k + j), Stay)) [0..<k])$   
*else*  
 $[(rs ! 0, Stay),$   
 $(rs ! 1, Left),$   
 $(rs ! 2, Stay)] @$   
 $(map (\lambda j. (rs ! (j + 3), Stay)) [0..<k]) @$   
 $(map (\lambda j. (rs ! (3 + k + j), Stay)) [0..<k])$

**abbreviation** *condition7a*  $gs\ jj \equiv$   
 $gs ! (3 + jj) = 0 \wedge enc-nth (gs ! 1) (k + jj) = 1 \wedge is-code (gs ! 1) \wedge \neg is-beginning (gs ! 1)$

**abbreviation** *condition7b*  $gs\ jj \equiv$   
 $\neg condition7a\ gs\ jj \wedge gs ! (3 + jj) = 3 \wedge is-code (gs ! 1)$

**abbreviation** *condition7c*  $gs\ jj \equiv$   
 $\neg condition7a\ gs\ jj \wedge \neg condition7b\ gs\ jj$

**lemma** *turing-command-cmdL7*:

**assumes**  $jj < k$

**shows** *turing-command*  $(2 * k + 3) 1 G' (cmdL7\ jj)$

*<proof>*

**definition** *tmL67*  $:: nat \Rightarrow machine$  **where**

$tmL67\ jj \equiv [cmdL7\ jj]$

**lemma** *tmL67-tm*:

**assumes**  $jj < k$

**shows** *turing-machine*  $(2 * k + 3) G' (tmL67\ jj)$

*<proof>*

**definition** *tmL47*  $jj \equiv tmL46\ jj ;; tmL67\ jj$

**lemma** *tmL47-tm*:

**assumes**  $jj < k$

**shows** *turing-machine*  $(2 * k + 3) G' (tmL47\ jj)$

*<proof>*

Next we are sweeping right again and perform the head movement for tape  $jj$  if this is a movement to the right. This works the same as the left movements in *cmdL7*.

**definition** *cmdL8*  $:: nat \Rightarrow command$  **where**

$cmdL8\ jj\ rs \equiv$

*if*  $rs ! 1 = \square$  *then* 1 *else* 0,

*if*  $rs ! (3 + jj) = 2 \wedge enc-nth (rs ! 1) (k + jj) = 1 \wedge is-code (rs ! 1)$  *then*

$[(rs ! 0, Stay),$

$(enc-upd (rs ! 1) (k + jj) 0, Right),$

$(rs ! 2, Stay)] @$

$(map (\lambda j. (if j = jj then 3 else rs ! (j + 3), Stay)) [0..<k]) @$

$(map (\lambda j. (rs ! (3 + k + j), Stay)) [0..<k])$

*else if*  $rs ! (3 + jj) = 3 \wedge is-code (rs ! 1)$  *then*

$[(rs ! 0, Stay),$

$(enc-upd (rs ! 1) (k + jj) 1, Right),$

$(rs ! 2, Stay)] @$

$(map (\lambda j. (if j = jj then 2 else rs ! (j + 3), Stay)) [0..<k]) @$

$(map (\lambda j. (rs ! (3 + k + j), Stay)) [0..<k])$

*else if*  $rs ! 1 = 0$  *then*

$[(rs ! 0, Stay),$

$(rs ! 1, Stay),$

$(rs ! 2, Stay)] @$

$(map (\lambda j. (rs ! (j + 3), Stay)) [0..<k]) @$

$(map (\lambda j. (rs ! (3 + k + j), Stay)) [0..<k])$

*else*

$[(rs ! 0, Stay),$

$(rs ! 1, Right),$   
 $(rs ! 2, Stay)] @$   
 $(map (\lambda j. (rs ! (j + 3), Stay)) [0..<k]) @$   
 $(map (\lambda j. (rs ! (3 + k + j), Stay)) [0..<k]))$

**abbreviation** *condition8a*  $gs\ jj \equiv$   
 $gs ! (3 + jj) = 2 \wedge enc\text{-}nth\ (gs ! 1)\ (k + jj) = 1 \wedge is\text{-}code\ (gs ! 1)$

**abbreviation** *condition8b*  $gs\ jj \equiv$   
 $\neg\ condition8a\ gs\ jj \wedge gs ! (3 + jj) = 3 \wedge is\text{-}code\ (gs ! 1)$

**abbreviation** *condition8c*  $gs\ jj \equiv$   
 $\neg\ condition8a\ gs\ jj \wedge \neg\ condition8b\ gs\ jj \wedge gs ! 1 = 0$

**abbreviation** *condition8d*  $gs\ jj \equiv$   
 $\neg\ condition8a\ gs\ jj \wedge \neg\ condition8b\ gs\ jj \wedge \neg\ condition8c\ gs\ jj$

**lemma** *turing-command-cmdL8*:

**assumes**  $jj < k$

**shows** *turing-command*  $(2 * k + 3)\ 1\ G'\ (cmdL8\ jj)$

*<proof>*

**definition** *tmL78*  $::\ nat \Rightarrow machine$  **where**

*tmL78*  $jj \equiv [cmdL8\ jj]$

**lemma** *tmL78-tm*:

**assumes**  $jj < k$

**shows** *turing-machine*  $(2 * k + 3)\ G'\ (tmL78\ jj)$

*<proof>*

**definition** *tmL48*  $jj \equiv tmL47\ jj\ ;;\ tmL78\ jj$

**lemma** *tmL48-tm*:

**assumes**  $jj < k$

**shows** *turing-machine*  $(2 * k + 3)\ G'\ (tmL48\ jj)$

*<proof>*

The last command in the command sequence is moving back to the beginning of the output tape.

**definition** *tmL49*  $jj \equiv tmL48\ jj\ ;;\ tm\text{-}left\text{-}until1$

The Turing machine *tmL49*  $jj$  is then repeated for the parameters  $jj = 0, \dots, k - 1$  in order to simulate the actions of  $M$  on all tapes.

**lemma** *tmL49-tm*:  $jj < k \implies turing\text{-}machine\ (2 * k + 3)\ G'\ (tmL49\ jj)$

*<proof>*

**fun** *tmL49-upt*  $::\ nat \Rightarrow machine$  **where**

*tmL49-upt*  $0 = []\ |$

*tmL49-upt*  $(Suc\ j) = tmL49\text{-}upt\ j\ ;;\ tmL49\ j$

**lemma** *tmL49-upt-tm*:

**assumes**  $j \leq k$

**shows** *turing-machine*  $(2 * k + 3)\ G'\ (tmL49\text{-}upt\ j)$

*<proof>*

**definition** *tmL9*  $\equiv tmL4\ ;;\ tmL49\text{-}upt\ k$

**lemma** *tmL9-tm*: *turing-machine*  $(2 * k + 3)\ G'\ tmL9$

*<proof>*

At this point in the iteration we have completed one more step in the execution of  $M$ . We mark this by setting one more counter flag, namely the one in the leftmost code symbol where the flag is still unset. To find the first unset counter flag, we reuse *tmC*.

**definition** *tmL10*  $\equiv tmL9\ ;;\ tmC$

**lemma** *tmL10-tm*: *turing-machine*  $(2 * k + 3)\ G'\ tmL10$

*<proof>*

Then we set the counter flag, unless we have reached a blank symbol.

**definition** *cmdL11* :: *command* **where**

```

cmdL11 rs ≡
  (1,
   [(rs ! 0, Stay),
    if is-code (rs ! 1) then (enc-upd (rs ! 1) (2 * k) 1, Stay) else (rs ! 1, Stay),
    (rs ! 2, Stay)] @
   (map (λj. (rs ! (j + 3), Stay)) [0..<k]) @
   (map (λj. (rs ! (3 + k + j), Stay)) [0..<k]))

```

**lemma** *turing-command-cmdL11*: *turing-command* (2 \* *k* + 3) 1 *G'* *cmdL11*  
 ⟨*proof*⟩

**definition** *tmL11* ≡ *tmL10* ;; [*cmdL11*]

**lemma** *tmL1011-tm*: *turing-machine* (2 \* *k* + 3) *G'* [*cmdL11*]  
 ⟨*proof*⟩

**lemma** *tmL11-tm*: *turing-machine* (2 \* *k* + 3) *G'* *tmL11*  
 ⟨*proof*⟩

And we move back to the beginning of the output tape again.

**definition** *tmL12* ≡ *tmL11* ;; *tm-left-until1*

**lemma** *tmL12-tm*: *turing-machine* (2 \* *k* + 3) *G'* *tmL12*  
 ⟨*proof*⟩

Now, at the end of the iteration we set the memorization tapes 3, ..., 2*k* + 2, that is, all but the one memorizing the state of *M*, to 0. This makes for a simpler loop invariant than having the leftover symbols there.

**definition** *tmL13* ≡ *tmL12* ;; *tm-write-many* {3..<2 \* *k* + 3} 0

**lemma** *tmL13-tm*: *turing-machine* (2 \* *k* + 3) *G'* *tmL13*  
 ⟨*proof*⟩

This is the entire loop. It terminates once there are no unset counter flags anymore.

**definition** *tmLoop* ≡ *WHILE* *tmC* ; λ*rs*. *rs* ! 1 > □ *DO* *tmL13* *DONE*

**lemma** *tmLoop-tm*: *turing-machine* (2 \* *k* + 3) *G'* *tmLoop*  
 ⟨*proof*⟩

**definition** *tm10* ≡ *tm9* ;; *tmLoop*

**lemma** *tm10-tm*: *turing-machine* (2 \* *k* + 3) *G'* *tm10*  
 ⟨*proof*⟩

## Cleaning up the output

Now the simulation proper has ended, but the output tape does not yet look quite like the output tape of *M*. It remains to extract the component 1 from all the code symbols on the output tape. The simulator does this while sweeping left. Formally, “extracting component 1” means this:

**abbreviation** *ec1* :: *symbol* ⇒ *symbol* **where**  
*ec1* *h* ≡ if *is-code* *h* then *enc-nth* *h* 1 else □

**lemma** *ec1*: *ec1* *h* < *G'* if *h* < *G'* for *h*  
 ⟨*proof*⟩

**definition** *tm11* ≡ *tm10* ;; *tm-ltrans-until* 1 1 *StartSym* *ec1*

**lemma** *tm11-tm*: *turing-machine* (2 \* *k* + 3) *G'* *tm11*  
 ⟨*proof*⟩

The previous TM, *tm-ltrans-until 1 1*  $\{y. y < G^2 * k + 2 + 2 \wedge 1 < y \wedge \text{dec } y ! (2 * k + 1) = 1\}$  ( $\lambda h.$  *if is-code h then enc-nth h 1 else 0*), halts as soon as it encounters a code symbol with the start flag set, without applying the extraction function. Applying the function to this final code symbol, which is at the leftmost cell of the tape, is the last step of the simulator machine.

**definition** *tm12*  $\equiv$  *tm11* ;; *tm-rtrans 1 ec1*

**lemma** *tm12-tm*: *turing-machine*  $(2 * k + 3)$  *G'* *tm12*  
*<proof>*

### 5.3.3 Semantics of the Turing machine

This section establishes not only the configurations of the simulator but also the traces. For every Turing machine and command defined in the previous subsection, there will be a corresponding trace and a tape list representing the simulator's configuration after the command or TM has been applied.

For most of the time, the simulator's output tape will have no start symbol, and so the next definition will be more suited to describing it than the customary *contents*.

**definition** *contents'* :: *symbol list*  $\Rightarrow$  (*nat*  $\Rightarrow$  *symbol*) **where**  
*contents' ys x*  $\equiv$  *if*  $x < \text{length } ys$  *then*  $ys ! x$  *else* 0

**lemma** *contents'-eqI*:  
**assumes**  $\bigwedge x. x < \text{length } ys \Longrightarrow zs\ x = ys ! x$   
**and**  $\bigwedge x. x \geq \text{length } ys \Longrightarrow zs\ x = 0$   
**shows**  $zs = \text{contents}'\ ys$   
*<proof>*

**lemma** *contents-contents'*:  $[ys] = \text{contents}'\ (1 \# ys)$   
*<proof>*

**lemma** *contents'-at-ge*:  
**assumes**  $i \geq \text{length } ys$   
**shows**  $\text{contents}'\ ys\ i = 0$   
*<proof>*

In the following context *zs* represents the input for *M* and hence for the simulator.

**context**  
**fixes** *zs* :: *symbol list*  
**assumes** *zs*: *bit-symbols zs*  
**begin**

**lemma** *zs-less-G*:  $\forall i < \text{length } zs. zs ! i < G$   
*<proof>*

**lemma** *zs-proper*: *proper-symbols zs*  
*<proof>*

**abbreviation** *n*  $\equiv$  *length zs*

**abbreviation** *TT*  $\equiv$  *Suc (fmt n)*

#### Initializing the simulator's tapes

**definition** *tps0* :: *tape list* **where**  
*tps0*  $\equiv$   
 $[[[zs], 0],$   
 $[\ [], 0]] @$   
*replicate*  $(2 * k + 1)$   $([\triangleright])$

**lemma** *tps0-start-config*: *start-config*  $(2 * k + 3)$  *zs* =  $(0, tps0)$   
*<proof>*

**definition** *tps1* :: *tape list* **where**

$tps1 \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lfloor replicate (fmt\ n)\ 3 \rfloor, 1)] @$   
 $replicate (2 * k + 1) (\lceil \triangleright \rceil)$

**definition**  $es1 \equiv es\text{-}fmt\ n$

**lemma**  $tm1$ : *traces*  $tm1\ tps0\ es1\ tps1$   
 $\langle proof \rangle$

**definition**  $es1\text{-}2 \equiv map (\lambda i. (1, 1 + Suc\ i)) [0..<fmt\ n] @ [(1, 1 + fmt\ n)]$

**definition**  $es2 \equiv es1 @ es1\text{-}2$

**lemma**  $len\text{-}es2$ :  $length\ es2 = length\ (es\text{-}fmt\ n) + TT$   
 $\langle proof \rangle$

**definition**  $tps2$  :: *tape list where*

$tps2 \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lfloor replicate (fmt\ n)\ (enc\ (replicate\ k\ 0 @ replicate\ k\ 0 @ [0, 0])) \rfloor, TT)] @$   
 $replicate (2 * k + 1) (\lceil \triangleright \rceil)$

**lemma**  $tm2$ : *traces*  $tm2\ tps0\ es2\ tps2$   
 $\langle proof \rangle$

**definition**  $es3 \equiv es2 @ map (\lambda i. (1, i)) (rev [0..<TT]) @ [(1, 0)]$

**definition**  $tps3$  :: *tape list where*

$tps3 \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(\lfloor replicate (fmt\ n)\ (enc\ (replicate\ k\ 0 @ replicate\ k\ 0 @ [0, 0])) \rfloor, 0)] @$   
 $replicate (2 * k + 1) (\lceil \triangleright \rceil)$

**lemma**  $tm3$ : *traces*  $tm3\ tps0\ es3\ tps3$   
 $\langle proof \rangle$

**definition**  $es4 \equiv es3 @ [(1, 0)]$

**lemma**  $len\text{-}es4$ :  $length\ es4 = length\ (es\text{-}fmt\ n) + 2 * TT + 2$   
 $\langle proof \rangle$

**definition**  $tps4$  :: *tape list where*

$tps4 \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(contents'$   
 $((enc\ (replicate\ k\ 1 @ replicate\ k\ 1 @ [0, 1])) \#$   
 $replicate\ (fmt\ n)\ (enc\ (replicate\ k\ 0 @ replicate\ k\ 0 @ [0, 0]))) \rfloor, 0)] @$   
 $replicate (2 * k + 1) (\lceil \triangleright \rceil)$

**lemma**  $tm4$ : *traces*  $tm4\ tps0\ es4\ tps4$   
 $\langle proof \rangle$

**definition**  $es5 \equiv es4 @ [(1, 1)]$

**lemma**  $len\text{-}es5$ :  $length\ es5 = length\ (es\text{-}fmt\ n) + 2 * TT + 3$   
 $\langle proof \rangle$

**definition**  $tps5$  :: *tape list where*

$tps5 \equiv$   
 $[(\lfloor zs \rfloor, 1),$   
 $(contents'$   
 $((enc\ (replicate\ k\ 1 @ replicate\ k\ 1 @ [0, 1])) \#$

*replicate (fmt n) (enc (replicate k 0 @ replicate k 0 @ [0, 0])), 1] @  
replicate (2 \* k + 1) ([▷])*

**lemma** *tm5: traces tm5 tps0 es5 tps5*  
⟨proof⟩

Since the simulator simulates  $M$  on  $zs$ , its tape contents are typically described in terms of configurations of  $M$ . So the following definition is actually more like an abbreviation.

**definition** *exec t*  $\equiv$  *execute M (start-config k zs) t*

**lemma** *exec-pos-less-TT*:  
**assumes**  $j < k$   
**shows** *exec t*  $<\#\>$   $j < TT$   
⟨proof⟩

**lemma** *tps-ge-TT-0*:  
**assumes**  $i \geq TT$   
**shows** (*exec t*  $<:\>$  1)  $i = 0$   
⟨proof⟩

The next definition is central to how we describe the simulator’s output tape. The basic idea is that it describes the tape during the simulation of the  $t$ -th step of executing  $M$  on input  $zs$ . Recall that  $TT$  is the length of the formatted part on the simulator’s output tape. The  $i$ -th cell of the output tape contains: (1)  $k$  symbols corresponding to the symbols in the  $i$ -th cell of the  $k$  tapes of  $M$  after  $t$  steps; (2)  $k$  flags indicating which of the  $k$  tape heads is in position  $i$ ; (3) a unary counter representing the number  $t$ ; (4) a flag indicating whether  $i = 0$ . This is the situation at the beginning of the  $t$ -iteration of the simulator’s main loop. During this iteration the tape changes slightly: some symbols and head positions may already represent the situation after  $t + 1$  steps of  $M$ , that is, the  $t$ -th step has been partially simulated already. To account for this, there is the  $xs$  parameter. It is meant to be set to a list of  $k$  pairs. Let the  $j$ -th element of this list be  $(a, b)$ . On  $M$ ’s tape  $j$  has already been simulated. In other words, the output tape reflects the situation after  $t + a$  steps. Likewise  $b$  will be either `None` or 0 or 1. If it is 0 or 1, it means the flag represents the head position of tape  $j$  after  $t + b$  steps. If it is `None`, it means that all head flags for tape  $k$  are currently zero, representing a “tape without head”. This situation occurs every time the simulator has cleared the head flag representing the position after  $t$  steps, but has not set the flag for the head position after  $t + 1$  steps yet.

Therefore, at the beginning of the  $t$ -iteration of the simulator’s loop  $xs$  consists of  $k$  pairs  $(0, 0)$ . During the iteration every pair will eventually become  $(0, 0)$ .

**definition** *zip-cont*  $::$   $nat \Rightarrow (nat \times nat\ option)\ list \Rightarrow (nat \Rightarrow symbol)$  **where**  
*zip-cont t xs i*  $\equiv$   
if  $i < TT$  then *enc*  
(*map* ( $\lambda j. (exec (t + fst (xs ! j)) <:\> j) i) [0..<k]$ ) @  
*map* ( $\lambda j. case\ snd\ (xs\ !\ j)\ of\ None \Rightarrow 0 \mid Some\ d \Rightarrow if\ i = exec\ (t + d)\ <\#\>\ j\ then\ 1\ else\ 0) [0..<k]$ ) @  
[if  $i < t$  then 1 else 0,  
if  $i = 0$  then 1 else 0])  
else 0

Some auxiliary lemmas for accessing elements of lists of certain shape:

**lemma** *less-k-nth*:  $j < k \implies (map\ f1\ [0..<k])\ @\ map\ f2\ [0..<k])\ @\ [a, b] ! j = f1\ j$   
⟨proof⟩

**lemma** *less-2k-nth*:  $k \leq j \implies j < 2 * k \implies (map\ f1\ [0..<k])\ @\ map\ f2\ [0..<k])\ @\ [a, b] ! j = f2\ (j - k)$   
⟨proof⟩

**lemma** *twok-nth*:  $(map\ f1\ [0..<k])\ @\ map\ f2\ [0..<k])\ @\ [a, b] ! (2 * k) = a$   
⟨proof⟩

**lemma** *twok1-nth*:  $(map\ f1\ [0..<k])\ @\ map\ f2\ [0..<k])\ @\ [a, b] ! (2 * k + 1) = b$   
⟨proof⟩

**lemma** *zip-cont-less-G*:  
**assumes**  $i < TT$

**shows**  $\forall x \in \text{set} (\text{map } (\lambda j. (\text{exec } (t + \text{fst } (xs ! j)) <:> j) i) [0..<k] @$   
 $\text{map } (\lambda j. \text{case } \text{snd } (xs ! j) \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } d \Rightarrow \text{if } i = \text{exec } (t + d) <\#\> j \text{ then } 1 \text{ else } 0) [0..<k] @$   
 $[\text{if } i < t \text{ then } 1 \text{ else } 0,$   
 $\text{if } i = 0 \text{ then } 1 \text{ else } 0]). x < G$   
**(is**  $\forall x \in \text{set} (?us @ ?vs @ [?a, ?b]). x < G$ )  
<proof>

**lemma** *dec-zip-cont:*

**assumes**  $i < TT$

**shows**  $\text{dec } (\text{zip-cont } t \text{ } xs \ i) =$

$(\text{map } (\lambda j. (\text{exec } (t + \text{fst } (xs ! j)) <:> j) i) [0..<k] @$   
 $\text{map } (\lambda j. \text{case } \text{snd } (xs ! j) \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } d \Rightarrow \text{if } i = \text{exec } (t + d) <\#\> j \text{ then } 1 \text{ else } 0) [0..<k] @$   
 $[\text{if } i < t \text{ then } 1 \text{ else } 0,$   
 $\text{if } i = 0 \text{ then } 1 \text{ else } 0])$   
**(is**  $- = ?ys$ )

<proof>

**lemma** *zip-cont-gt-1:*

**assumes**  $i < TT$

**shows**  $\text{zip-cont } t \text{ } xs \ i > 1$

<proof>

**lemma** *zip-cont-less:*

**assumes**  $i < TT$

**shows**  $\text{zip-cont } t \text{ } xs \ i < G \wedge (2 * k + 2) + 2$

<proof>

**lemma** *zip-cont-eq-0:*

**assumes**  $i \geq TT$

**shows**  $\text{zip-cont } t \text{ } xs \ i = 0$

<proof>

**lemma** *dec-zip-cont-less-k:*

**assumes**  $i < TT$  **and**  $j < k$

**shows**  $\text{dec } (\text{zip-cont } t \text{ } xs \ i) ! j = (\text{exec } (t + \text{fst } (xs ! j)) <:> j) i$

<proof>

**lemma** *dec-zip-cont-less-2k:*

**assumes**  $i < TT$  **and**  $j \geq k$  **and**  $j < 2 * k$

**shows**  $\text{dec } (\text{zip-cont } t \text{ } xs \ i) ! j =$

$(\text{case } \text{snd } (xs ! (j - k)) \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } d \Rightarrow \text{if } i = \text{exec } (t + d) <\#\> (j - k) \text{ then } 1 \text{ else } 0)$

<proof>

**lemma** *dec-zip-cont-2k:*

**assumes**  $i < TT$

**shows**  $\text{dec } (\text{zip-cont } t \text{ } xs \ i) ! (2 * k) = (\text{if } i < t \text{ then } 1 \text{ else } 0)$

<proof>

**lemma** *dec-zip-cont-2k1:*

**assumes**  $i < TT$

**shows**  $\text{dec } (\text{zip-cont } t \text{ } xs \ i) ! (2 * k + 1) = (\text{if } i = 0 \text{ then } 1 \text{ else } 0)$

<proof>

**lemma** *zip-cont-eqI:*

**assumes**  $i < TT$

**and**  $\bigwedge j. j < k \implies (\text{exec } (t + \text{fst } (xs ! j)) <:> j) i = (\text{exec } (t + \text{fst } (xs' ! j)) <:> j) i$

**and**  $\bigwedge j. j < k \implies$

$(\text{case } \text{snd } (xs ! j) \text{ of } \text{None} \Rightarrow (0::\text{nat}) \mid \text{Some } d \Rightarrow \text{if } i = \text{exec } (t + d) <\#\> j \text{ then } 1 \text{ else } 0) =$

$(\text{case } \text{snd } (xs' ! j) \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } d \Rightarrow \text{if } i = \text{exec } (t + d) <\#\> j \text{ then } 1 \text{ else } 0)$

**shows**  $\text{zip-cont } t \text{ } xs \ i = \text{zip-cont } t \text{ } xs' \ i$

<proof>

**lemma** *zip-cont-nth-eqI:*

**assumes**  $i < TT$   
**and**  $\bigwedge j. j < k \implies (\text{exec } (t + \text{fst } (xs ! j)) <:> j) \ i = (\text{exec } (t + \text{fst } (xs' ! j)) <:> j) \ i$   
**and**  $\bigwedge j. j < k \implies \text{snd } (xs ! j) = \text{snd } (xs' ! j)$   
**shows**  $\text{zip-cont } t \ xs \ i = \text{zip-cont } t \ xs' \ i$   
 $\langle \text{proof} \rangle$

**lemma** *begin-tape-zip-cont*:  
 $\text{begin-tape } \{y. y < G \wedge (2 * k + 2) + 2 \wedge 1 < y \wedge \text{dec } y ! (2 * k + 1) = 1\} \ (\text{zip-cont } t \ xs, i)$   
 $(\text{is begin-tape } ?ys \ -)$   
 $\langle \text{proof} \rangle$

**definition**  $es6 \equiv es5 \ @ \ \text{map } (\lambda i. (1 + \text{Suc } i, 1 + \text{Suc } i)) \ [0..<n] \ @ \ [(1 + n, 1 + n)]$

**lemma** *len-es6*:  $\text{length } es6 = \text{length } (es\text{-fmt } n) + 2 * TT + n + 4$   
 $\langle \text{proof} \rangle$

**definition** *tps6* :: *tape list where*  
 $tps6 \equiv$   
 $[[[zs], n + 1],$   
 $(\text{zip-cont } 0 \ (\text{replicate } k \ (0, \text{Some } 0)), n + 1)] \ @$   
 $\text{replicate } (2 * k + 1) \ ([\triangleright])$

**lemma** *tm6*:  $\text{traces } tm6 \ tps0 \ es6 \ tps6$   
 $\langle \text{proof} \rangle$

**definition** *tps7* :: *tape list where*  
 $tps7 \equiv$   
 $[[[zs], n + 1],$   
 $(\text{zip-cont } 0 \ (\text{replicate } k \ (0, \text{Some } 0)), 0)] \ @$   
 $\text{replicate } (2 * k + 1) \ ([\triangleright])$

**definition**  $es7 \equiv es6 \ @ \ \text{map } (\lambda i. (n + 1, i)) \ (\text{rev } [0..<n + 1]) \ @ \ [(n + 1, 0)]$

**lemma** *len-es7*:  $\text{length } es7 = \text{length } (es\text{-fmt } n) + 2 * TT + 2 * n + 6$   
 $\langle \text{proof} \rangle$

**lemma** *tm7*:  $\text{traces } tm7 \ tps0 \ es7 \ tps7$   
 $\langle \text{proof} \rangle$

**definition** *tps8* :: *tape list where*  
 $tps8 \equiv$   
 $[[[zs], n + 1],$   
 $(\text{zip-cont } 0 \ (\text{replicate } k \ (0, \text{Some } 0)), 0),$   
 $[\square]] \ @$   
 $\text{replicate } (2 * k) \ ([\triangleright])$

**definition**  $es8 \equiv es7 \ @ \ [(n + 1, 0)]$

**lemma** *len-es8*:  $\text{length } es8 = \text{length } (es\text{-fmt } n) + 2 * TT + 2 * n + 7$   
 $\langle \text{proof} \rangle$

**lemma** *tm8*:  $\text{traces } tm8 \ tps0 \ es8 \ tps8$   
 $\langle \text{proof} \rangle$

**definition** *tps9* :: *tape list where*  
 $tps9 \equiv$   
 $[[[zs], n + 1],$   
 $(\text{zip-cont } 0 \ (\text{replicate } k \ (0, \text{Some } 0)), 0),$   
 $[\square]] \ @$   
 $\text{replicate } (2 * k) \ ([\square])$

**definition**  $es9 \equiv es8 \ @ \ [(n + 1, 0)]$

**lemma** *len-es9*:  $\text{length } es9 = \text{length } (es\text{-fmt } n) + 2 * TT + 2 * n + 8$   
 ⟨proof⟩

**lemma** *tm9*:  $\text{traces } tm9 \text{ tps0 } es9 \text{ tps9}$   
 ⟨proof⟩

## The loop

Immediately before and during the loop the tapes will have the shape below. The input tape will stay unchanged. The output tape will contain the  $k$  encoded tapes of  $M$ . The first memorization tape will contain  $M$ 's state. The following  $k$  memorization tapes will store information about head movements. The final  $k$  memorization tapes will have information about read or to-be-written symbols.

**definition** *tpsL* ::  $\text{nat} \Rightarrow (\text{nat} \times \text{nat option}) \text{ list} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow (\text{nat} \Rightarrow \text{symbol}) \Rightarrow \text{tape list}$   
 where

$\text{tpsL } t \text{ xs } i \text{ q mvs syms} \equiv$   
 ( $\lfloor \text{zs} \rfloor, n + 1$ ) #  
 ( $\text{zip-cont } t \text{ xs}, i$ ) #  
 [ $\text{fst } (\text{exec } (t + q))$ ] #  
 $\text{map } (\text{onesie} \circ \text{mvs}) [0..<k]$  @  
 $\text{map } (\text{onesie} \circ \text{syms}) [0..<k]$

**lemma** *length-tpsL* [*simp*]:  $\text{length } (\text{tpsL } t \text{ xs } i \text{ q mvs syms}) = 2 * k + 3$   
 ⟨proof⟩

**lemma** *tpsL-pos-0*:  $\text{tpsL } t \text{ xs } i \text{ q mvs syms} : \# : 0 = n + 1$   
 ⟨proof⟩

**lemma** *tpsL-pos-1*:  $\text{tpsL } t \text{ xs } i \text{ q mvs syms} : \# : 1 = i$   
 ⟨proof⟩

**lemma** *read-tpsL-0*:  $\text{read } (\text{tpsL } t \text{ xs } i \text{ q mvs syms}) ! 0 = \square$   
 ⟨proof⟩

**lemma** *read-tpsL-1*:  $\text{read } (\text{tpsL } t \text{ xs } i \text{ q mvs syms}) ! 1 =$   
 ( $\text{if } i < TT \text{ then enc}$   
 ( $\text{map } (\lambda j. (\text{exec } (t + \text{fst } (xs ! j)) <:> j) i) [0..<k]$  @  
 $\text{map } (\lambda j. \text{case snd } (xs ! j) \text{ of None} \Rightarrow 0 \mid \text{Some } d \Rightarrow \text{if } i = \text{exec } (t + d) <\#\#> j \text{ then } 1 \text{ else } 0) [0..<k]$  @  
 [ $\text{if } i < t \text{ then } 1 \text{ else } 0,$   
 $\text{if } i = 0 \text{ then } 1 \text{ else } 0]$ )  
 else 0)  
 ⟨proof⟩

**lemma** *read-tpsL-1-nth-less-k*:  
**assumes**  $i < TT$  **and**  $j < k$   
**shows**  $\text{enc-nth } (\text{read } (\text{tpsL } t \text{ xs } i \text{ q mvs syms}) ! 1) j = (\text{exec } (t + \text{fst } (xs ! j)) <:> j) i$   
 ⟨proof⟩

**lemma** *read-tpsL-1-nth-less-2k*:  
**assumes**  $i < TT$  **and**  $k \leq j$  **and**  $j < 2 * k$   
**shows**  $\text{enc-nth } (\text{read } (\text{tpsL } t \text{ xs } i \text{ q mvs syms}) ! 1) j =$   
 ( $\text{case snd } (xs ! (j - k)) \text{ of None} \Rightarrow 0 \mid \text{Some } d \Rightarrow \text{if } i = \text{exec } (t + d) <\#\#> (j - k) \text{ then } 1 \text{ else } 0$ )  
 ⟨proof⟩

**lemma** *read-tpsL-1-nth-2k*:  
**assumes**  $i < TT$   
**shows**  $\text{enc-nth } (\text{read } (\text{tpsL } t \text{ xs } i \text{ q mvs syms}) ! 1) (2 * k) = (\text{if } i < t \text{ then } 1 \text{ else } 0)$   
 ⟨proof⟩

**lemma** *read-tpsL-1-nth-2k1*:  
**assumes**  $i < TT$   
**shows**  $\text{enc-nth } (\text{read } (\text{tpsL } t \text{ xs } i \text{ q mvs syms}) ! 1) (2 * k + 1) = (\text{if } i = 0 \text{ then } 1 \text{ else } 0)$   
 ⟨proof⟩

**lemma** *read-tpsL-1-bounds*:  
**assumes**  $i < TT$   
**shows**  $\text{read } (tpsL\ t\ xs\ i\ q\ mvs\ syms) ! 1 > 1$   
**and**  $\text{read } (tpsL\ t\ xs\ i\ q\ mvs\ syms) ! 1 < G \wedge (2 * k + 2) + 2$   
 $\langle \text{proof} \rangle$

**lemma** *read-tpsL-2*:  $\text{read } (tpsL\ t\ xs\ i\ q\ mvs\ syms) ! 2 = \text{fst } (\text{exec } (t + q))$   
 $\langle \text{proof} \rangle$

**lemma** *read-tpsL-3*:  
**assumes**  $\beta \leq j$  **and**  $j < k + 3$   
**shows**  $\text{read } (tpsL\ t\ xs\ i\ q\ mvs\ syms) ! j = mvs\ (j - \beta)$   
 $\langle \text{proof} \rangle$

**lemma** *read-tpsL-3'*:  
**assumes**  $j < k$   
**shows**  $\text{read } (tpsL\ t\ xs\ i\ q\ mvs\ syms) ! (j + \beta) = mvs\ j$   
 $\langle \text{proof} \rangle$

**lemma** *read-tpsL-4*:  
**assumes**  $k + \beta \leq j$  **and**  $j < 2 * k + \beta$   
**shows**  $\text{read } (tpsL\ t\ xs\ i\ q\ mvs\ syms) ! j = \text{syms } (j - k - \beta)$   
 $\langle \text{proof} \rangle$

**lemma** *map-const-upt*:  $\text{map } (\text{onesie } \circ (\lambda \cdot. c))\ [0..<k] = \text{replicate } k\ [c]$   
 $\langle \text{proof} \rangle$

After the initialization, that is, right before the loop, the simulator's tapes look like this:

**lemma** *tps9-tpsL*:  $\text{tps9} = \text{tpsL } 0\ (\text{replicate } k\ (0, \text{Some } 0))\ 0\ 0\ (\lambda j. 0)\ (\lambda j. 0)$   
 $\langle \text{proof} \rangle$

**lemma** *tpsL-0*:  $\text{tpsL } t\ xs\ i\ q\ mvs\ syms ! 0 = (\lfloor zs \rfloor, n + 1)$   
 $\langle \text{proof} \rangle$

**lemma** *tpsL-1*:  $\text{tpsL } t\ xs\ i\ q\ mvs\ syms ! 1 = (\text{zip-cont } t\ xs, i)$   
 $\langle \text{proof} \rangle$

**lemma** *tpsL-2*:  $\text{tpsL } t\ xs\ i\ q\ mvs\ syms ! 2 = \lceil \text{fst } (\text{exec } (t + q)) \rceil$   
 $\langle \text{proof} \rangle$

**lemma** *tpsL-mvs*:  $j < k \implies \text{tpsL } t\ xs\ i\ q\ mvs\ syms ! (\beta + j) = \lceil mvs\ j \rceil$   
 $\langle \text{proof} \rangle$

**lemma** *tpsL-mvs'*:  $\beta \leq j \implies j < \beta + k \implies \text{tpsL } t\ xs\ i\ q\ mvs\ syms ! j = \lceil mvs\ (j - \beta) \rceil$   
 $\langle \text{proof} \rangle$

**lemma** *tpsL-syms*:  
**assumes**  $j < k$   
**shows**  $\text{tpsL } t\ xs\ i\ q\ mvs\ syms ! (\beta + k + j) = \lceil \text{syms } j \rceil$   
 $\langle \text{proof} \rangle$

**lemma** *tpsL-syms'*:  
**assumes**  $\beta + k \leq j$  **and**  $j < 2 * k + \beta$   
**shows**  $\text{tpsL } t\ xs\ i\ q\ mvs\ syms ! j = \lceil \text{syms } (j - k - \beta) \rceil$   
 $\langle \text{proof} \rangle$

The condition: less than  $TT$  steps simulated.

**definition** *tpsC0* ::  $\text{nat} \Rightarrow \text{tape list}$  **where**  
 $\text{tpsC0 } t \equiv \text{tpsL } t\ (\text{replicate } k\ (0, \text{Some } 0))\ 0\ 0\ (\lambda j. 0)\ (\lambda j. 0)$

**definition** *tpsC1*  $t \equiv \text{tpsL } t\ (\text{replicate } k\ (0, \text{Some } 0))\ t\ 0\ (\lambda j. 0)\ (\lambda j. 0)$

**definition** *esC*  $t \equiv \text{map } (\lambda i. (n + 1, \text{Suc } i))\ [0..<t] @ [(n + 1, t)]$

**lemma** *set-filter-upt*:  $x \in \text{set } (\text{filter } f [0..<N]) \implies x < N$   
 ⟨proof⟩

**lemma** *set-filter-upt'*:  $x \in \text{set } (\text{filter } f [0..<N]) \implies f x$   
 ⟨proof⟩

We will use this TM at the end of the loop again. Therefore we prove a more general result than necessary at this point.

**lemma** *tmC-general*:  
**assumes**  $t \leq TT$   
**and**  $\text{tps} = \text{tpsL } t \text{ xs } 0 \text{ i mvs syms}$   
**and**  $\text{tps}' = \text{tpsL } t \text{ xs } t \text{ i mvs syms}$   
**shows**  $\text{traces tmC tps } (\text{esC } t) \text{ tps}'$   
 ⟨proof⟩

**corollary** *tmC*:  
**assumes**  $t \leq TT$   
**shows**  $\text{traces tmC } (\text{tpsC0 } t) (\text{esC } t) (\text{tpsC1 } t)$   
 ⟨proof⟩

**lemma** *tpsC1-at-T*:  $\text{tpsC1 } TT \text{ :: } 1 = 0$   
 ⟨proof⟩

**lemma** *tpsC1-at-less-T*:  $t < TT \implies \text{tpsC1 } t \text{ :: } 1 > 0$   
 ⟨proof⟩

The body of the loop: simulating one step

**definition** *tpsL0*  $t \equiv \text{tpsL } t (\text{replicate } k (0, \text{Some } 0)) t 0 (\lambda j. 0) (\lambda j. 0)$

**lemma** *tpsL0-eq-tpsC1*:  $\text{tpsL0 } t = \text{tpsC1 } t$   
 ⟨proof⟩

**definition** *esL1*  $t \equiv \text{map } (\lambda i. (n + 1, i)) (\text{rev } [0..<t]) @ [(n + 1, 0)]$

**definition** *tpsL1*  $t \equiv \text{tpsL } t (\text{replicate } k (0, \text{Some } 0)) 0 0 (\lambda j. 0) (\lambda j. 0)$

**lemma** *tmL1*:  $\text{traces tmL1 } (\text{tpsL0 } t) (\text{esL1 } t) (\text{tpsL1 } t)$   
 ⟨proof⟩

Collecting the read symbols of the simulated machines:

**lemma** *sem-cmdL2-ge-TT*:  
**assumes**  $\text{tps} = \text{tpsL } t \text{ xs } i \text{ q mvs syms}$  **and**  $i \geq TT$   
**shows**  $\text{sem cmdL2 } (0, \text{tps}) = (1, \text{tps})$   
 ⟨proof⟩

**lemma** *sem-cmdL2-less-TT*:  
**assumes**  $\text{tps} = \text{tpsL } t \text{ xs } i \text{ q mvs syms}$   
**and**  $\text{syms} = (\lambda j. \text{if exec } t <\#\#> j < i \text{ then exec } t <.> j \text{ else } 0)$   
**and**  $\text{tps}' = \text{tpsL } t \text{ xs } (\text{Suc } i) \text{ q mvs syms}'$   
**and**  $\text{syms}' = (\lambda j. \text{if exec } t <\#\#> j < \text{Suc } i \text{ then exec } t <.> j \text{ else } 0)$   
**and**  $i < TT$   
**and**  $\text{xs} = \text{replicate } k (0, \text{Some } 0)$   
**shows**  $\text{sem cmdL2 } (0, \text{tps}) = (0, \text{tps}')$   
 ⟨proof⟩

**corollary** *sem-cmdL2-less-Tfmt*:  
**assumes**  $\text{xs} = \text{replicate } k (0, \text{Some } 0)$  **and**  $i < TT$   
**shows**  $\text{sem cmdL2}$   
 $(0, \text{tpsL } t \text{ xs } i \text{ q mvs } (\lambda j. \text{if exec } t <\#\#> j < i \text{ then exec } t <.> j \text{ else } \square)) =$   
 $(0, \text{tpsL } t \text{ xs } (\text{Suc } i) \text{ q mvs } (\lambda j. \text{if exec } t <\#\#> j < \text{Suc } i \text{ then exec } t <.> j \text{ else } \square))$   
 ⟨proof⟩

**lemma** *execute-cmdL2-le-TT*:

**assumes**  $tt \leq TT$  **and**  $xs = \text{replicate } k \ (0, \text{Some } 0)$  **and**  $tps = \text{tpsL } t \ xs \ 0 \ q \ mvs \ (\lambda\cdot. \square)$   
**shows**  $\text{execute } tmL1-2 \ (0, tps) \ tt =$   
 $(0, \text{tpsL } t \ xs \ tt \ q \ mvs \ (\lambda j. \text{if } \text{exec } t \ <\#\> \ j < \ tt \ \text{then } \text{exec } t \ <\cdot\> \ j \ \text{else } \square))$   
 $\langle \text{proof} \rangle$

**lemma** *tpsL-symsb-eq*:

**assumes**  $\bigwedge j. j < k \implies \text{symsb } j = \text{symsb}' j$   
**shows**  $\text{tpsL } t \ xs \ i \ q \ mvs \ \text{symsb} = \text{tpsL } t \ xs \ i \ q \ mvs \ \text{symsb}'$   
 $\langle \text{proof} \rangle$

**lemma** *execute-cmdL2-Suc-TT*:

**assumes**  $xs = \text{replicate } k \ (0, \text{Some } 0)$  **and**  $tps = \text{tpsL } t \ xs \ 0 \ q \ mvs \ (\lambda j. 0)$  **and**  $t < TT$   
**shows**  $\text{execute } tmL1-2 \ (0, tps) \ (\text{Suc } TT) = (1, \text{tpsL } t \ xs \ TT \ q \ mvs \ (\lambda j. \text{exec } t \ <\cdot\> \ j))$   
 $\langle \text{proof} \rangle$

**definition**  $esL1-2 \equiv \text{map } (\lambda i. (n + 1, \text{Suc } i)) \ [0..<TT] \ @ \ [(n + 1, TT)]$

**lemma** *traces-tmL1-2*:

**assumes**  $xs = \text{replicate } k \ (0, \text{Some } 0)$  **and**  $t < TT$   
**shows**  $\text{traces } tmL1-2 \ (\text{tpsL } t \ xs \ 0 \ q \ mvs \ (\lambda\cdot. \square)) \ esL1-2 \ (\text{tpsL } t \ xs \ TT \ q \ mvs \ (\lambda j. \text{exec } t \ <\cdot\> \ j))$   
 $\langle \text{proof} \rangle$

**definition**  $esL2 \ t \equiv esL1 \ t \ @ \ esL1-2$

**definition**  $tpsL2 \ t \equiv \text{tpsL } t \ (\text{replicate } k \ (0, \text{Some } 0)) \ TT \ 0 \ (\lambda\cdot. \square) \ (\lambda j. \text{exec } t \ <\cdot\> \ j)$

**lemma** *tmL2*:

**assumes**  $t < TT$   
**shows**  $\text{traces } tmL2 \ (\text{tpsL } 0 \ t) \ (esL2 \ t) \ (tpsL2 \ t)$   
 $\langle \text{proof} \rangle$

**definition**  $sim\text{-nextstate } t \equiv$

$(\text{if } \text{fst } (\text{exec } t) < \text{length } M$   
 $\text{then } \text{fst } ((M ! (\text{fst } (\text{exec } t))) \ (\text{config-read } (\text{exec } t)))$   
 $\text{else } \text{fst } (\text{exec } t))$

**lemma** *sim-nextstate*:  $\text{fst } (\text{exec } (\text{Suc } t)) = \text{sim-nextstate } t$

$\langle \text{proof} \rangle$

**definition**  $sim\text{-write } t \equiv$

$(\text{if } \text{fst } (\text{exec } t) < \text{length } M$   
 $\text{then } \text{map } \text{fst } (\text{snd } ((M ! (\text{fst } (\text{exec } t))) \ (\text{config-read } (\text{exec } t))))$   
 $\text{else } \text{config-read } (\text{exec } t))$

**lemma** *sim-write-nth*:

**assumes**  $\text{fst } (\text{exec } t) < \text{length } M$  **and**  $j < k$   
**shows**  $\text{sim-write } t \ ! \ j = ((M ! (\text{fst } (\text{exec } t))) \ (\text{config-read } (\text{exec } t)) \ [.] \ j)$   
 $\langle \text{proof} \rangle$

**lemma** *sim-write-nth-else*:

**assumes**  $\neg (\text{fst } (\text{exec } t) < \text{length } M)$   
**shows**  $\text{sim-write } t \ ! \ j = \text{config-read } (\text{exec } t) \ ! \ j$   
 $\langle \text{proof} \rangle$

**lemma** *sim-write-nth-less-G*:

**assumes**  $j < k$   
**shows**  $\text{sim-write } t \ ! \ j < G$   
 $\langle \text{proof} \rangle$

**lemma** *sim-write*:

**assumes**  $j < k$   
**shows**  $\text{exec } (\text{Suc } t) \ <\cdot\> \ j = (\text{exec } t \ <\cdot\> \ j)(\text{exec } t \ <\#\> \ j := \text{sim-write } t \ ! \ j)$

*<proof>*

**corollary** *sim-write'*:

**assumes**  $j < k$

**shows**  $(\text{exec } (\text{Suc } t) <:> j) (\text{exec } t <\#\> j) = \text{sim-write } t ! j$

*<proof>*

**definition** *sim-move*  $t \equiv \text{map direction-to-symbol}$

(if  $\text{fst } (\text{exec } t) < \text{length } M$

then  $\text{map snd } (\text{snd } ((M ! (\text{fst } (\text{exec } t))) (\text{config-read } (\text{exec } t))))$

else  $\text{replicate } k \text{ Stay}$ )

**lemma** *sim-move-nth*:

**assumes**  $\text{fst } (\text{exec } t) < \text{length } M$  **and**  $j < k$

**shows**  $\text{sim-move } t ! j = \text{direction-to-symbol } ((M ! (\text{fst } (\text{exec } t))) (\text{config-read } (\text{exec } t)) [\sim] j)$

*<proof>*

**lemma** *sim-move-nth-else*:

**assumes**  $\neg (\text{fst } (\text{exec } t) < \text{length } M)$  **and**  $j < k$

**shows**  $\text{sim-move } t ! j = 1$

*<proof>*

**lemma** *sim-move*:

**assumes**  $j < k$

**shows**  $\text{exec } (\text{Suc } t) <\#\> j = \text{exec } t <\#\> j + \text{sim-move } t ! j - 1$

*<proof>*

**definition** *tpsL3*  $t \equiv \text{tpsL}$

$t$

( $\text{replicate } k (0, \text{Some } 0)$ )

$TT$

$1$

( $\lambda j. \text{sim-move } t ! j$ )

( $\lambda j. \text{sim-write } t ! j$ )

**lemma** *read-execute*:  $\text{config-read } (\text{exec } t) = \text{map } (\lambda j. (\text{exec } t) <.> j) [0..<k]$

(**is**  $?lhs = ?rhs$ )

*<proof>*

**lemma** *sem-cmdL3*:  $\text{sem cmdL3 } (0, \text{tpsL2 } t) = (1, \text{tpsL3 } t)$

*<proof>*

**lemma** *execute-tmL2-3*:  $\text{execute tmL2-3 } (0, \text{tpsL2 } t) 1 = (1, \text{tpsL3 } t)$

*<proof>*

**definition** *esL3*  $t \equiv \text{esL2 } t @ [(n + 1, TT)]$

**lemma** *tmL3*:

**assumes**  $t < TT$

**shows**  $\text{traces tmL3 } (\text{tpsL0 } t) (\text{esL3 } t) (\text{tpsL3 } t)$

*<proof>*

**definition** *esL4*  $t \equiv \text{esL3 } t @ \text{map } (\lambda i. (n + 1, i)) (\text{rev } [0..<TT]) @ [(n + 1, 0)]$

**lemma** *len-esL4*:  $\text{length } (\text{esL4 } t) = t + 2 * TT + 4$

*<proof>*

**definition** *tpsL4*  $t \equiv \text{tpsL}$

$t$

( $\text{replicate } k (0, \text{Some } 0)$ )

$0$

$1$

( $\lambda j. \text{sim-move } t ! j$ )

( $\lambda j$ . *sim-write*  $t ! j$ )

**lemma** *tmL4*:  
 **assumes**  $t < TT$   
 **shows** *traces* *tmL4* (*tpsL0*  $t$ ) (*esL4*  $t$ ) (*tpsL4*  $t$ )  
 *<proof>*

**lemma** *enc-upd-zip-cont-None-Some*:  
 **assumes**  $jj < k$   
 **and** *length*  $xs = k$   
 **and**  $xs ! jj = (1, None)$   
 **and**  $i = (\text{exec } (Suc\ t) <\#\#> jj)$   
 **shows** *enc-upd* (*zip-cont*  $t\ xs\ i$ ) ( $k + jj$ )  $1 = \text{zip-cont } t\ (xs[jj:= (1, Some\ 1)])\ i$   
 *<proof>*

**lemma** *enc-upd-zip-cont-None-Some-Right*:  
 **assumes**  $jj < k$   
 **and** *length*  $xs = k$   
 **and**  $xs ! jj = (1, None)$   
 **and**  $i = Suc\ (\text{exec } t <\#\#> jj)$   
 **and** *sim-move*  $t ! jj = 2$   
 **shows** *enc-upd* (*zip-cont*  $t\ xs\ i$ ) ( $k + jj$ )  $1 = \text{zip-cont } t\ (xs[jj:= (1, Some\ 1)])\ i$   
 *<proof>*

**lemma** *enc-upd-zip-cont-None-Some-Left*:  
 **assumes**  $jj < k$   
 **and** *length*  $xs = k$   
 **and**  $xs ! jj = (1, None)$   
 **and**  $Suc\ i = \text{exec } t <\#\#> jj$   
 **and** *sim-move*  $t ! jj = 0$   
 **shows** *enc-upd* (*zip-cont*  $t\ xs\ i$ ) ( $k + jj$ )  $1 = \text{zip-cont } t\ (xs[jj:= (1, Some\ 1)])\ i$   
 *<proof>*

**lemma** *enc-upd-zip-cont-Some-None*:  
 **assumes**  $jj < k$   
 **and** *length*  $xs = k$   
 **and**  $xs ! jj = (1, Some\ 0)$   
 **and**  $i = \text{exec } t <\#\#> jj$   
 **shows** *enc-upd* (*zip-cont*  $t\ xs\ i$ ) ( $k + jj$ )  $0 = \text{zip-cont } t\ (xs[jj:= (1, None)])\ i$   
 *<proof>*

**lemma** *zip-cont-nth-eq-updI1*:  
 **assumes**  $i < TT$   
 **and**  $jj < k$   
 **and** *length*  $xs = k$   
 **and**  $xs ! jj = (0, Some\ 0)$   
 **and**  $(\text{exec } (Suc\ t) <:\#> jj)\ i = u$   
 **shows** *enc-upd* (*zip-cont*  $t\ xs\ i$ )  $jj\ u = \text{zip-cont } t\ (xs[jj:= (1, Some\ 0)])\ i$   
 *<proof>*

**lemma** *zip-cont-upd-eq*:  
 **assumes**  $jj < k$   
 **and**  $i = \text{exec } t <\#\#> jj$   
 **and**  $i < TT$   
 **and**  $xs ! jj = (0, Some\ 0)$   
 **and** *length*  $xs = k$   
 **shows** (*zip-cont*  $t\ xs$ )( $i := \text{enc-upd } (\text{zip-cont } t\ xs\ i)\ jj\ (\text{sim-write } t ! jj)$ ) =  
 *zip-cont*  $t\ (xs[jj:= (1, Some\ 0)])$   
 (**is** ?lhs = ?rhs)  
 *<proof>*

**lemma** *sem-cmdL5-neq-pos*:  
 **assumes**  $jj < k$

**and**  $tps = tpsL\ t\ xs\ i\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $snd\ (xs!\ jj) = Some\ 0$   
**and**  $i \neq exec\ t\ <\#\>\ jj$   
**and**  $i < TT$   
**and**  $tps' = tpsL\ t\ xs\ (Suc\ i)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $sem\ (cmdL5\ jj)\ (0, tps) = (0, tps')$   
*<proof>*

**lemma** *sem-cmdL5-eq-pos:*

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ i\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $xs!\ jj = (0, Some\ 0)$   
**and**  $i = exec\ t\ <\#\>\ jj$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 0)])\ (Suc\ i)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**shows**  $sem\ (cmdL5\ jj)\ (0, tps) = (0, tps')$   
*<proof>*

**lemma** *sem-cmdL5-eq-TT:*

**assumes**  $jj < k$  **and**  $tps = tpsL\ t\ xs\ TT\ q\ mvs\ syms$   
**shows**  $sem\ (cmdL5\ jj)\ (0, tps) = (1, tps)$   
*<proof>*

**lemma** *execute-tmL45-1:*

**assumes**  $tt \leq exec\ t\ <\#\>\ jj$   
**and**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $xs!\ jj = (0, Some\ 0)$   
**and**  $tps' = tpsL\ t\ xs\ tt\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $execute\ (tmL45\ jj)\ (0, tps)\ tt = (0, tps')$   
*<proof>*

**lemma** *execute-tmL45-2:*

**assumes**  $tt = exec\ t\ <\#\>\ jj$   
**and**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $xs!\ jj = (0, Some\ 0)$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 0)])\ (Suc\ tt)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**shows**  $execute\ (tmL45\ jj)\ (0, tps)\ (Suc\ tt) = (0, tps')$   
*<proof>*

**lemma** *execute-tmL45-3:*

**assumes**  $tt \geq Suc\ (exec\ t\ <\#\>\ jj)$   
**and**  $tt \leq TT$   
**and**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $xs!\ jj = (0, Some\ 0)$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 0)])\ tt\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**shows**  $execute\ (tmL45\ jj)\ (0, tps)\ tt = (0, tps')$   
*<proof>*

**lemma** *execute-tmL45-4:*

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $xs!\ jj = (0, Some\ 0)$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 0)])\ TT\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**shows**  $execute\ (tmL45\ jj)\ (0, tps)\ (Suc\ TT) = (1, tps')$   
*<proof>*

**definition**  $esL45 \equiv map\ (\lambda i. (n + 1, Suc\ i))\ [0..<TT]\ @\ [(n + 1, TT)]$

**lemma** *len-esL45*:  $\text{length } esL45 = \text{Suc } TT$   
 ⟨*proof*⟩

**lemma** *nth-map-upt-TT*:  
**fixes**  $es$   
**assumes**  $es = \text{map } f [0..<TT] @ es'$  **and**  $i < TT$   
**shows**  $es ! i = f i$   
 ⟨*proof*⟩

**lemma** *tmL45*:  
**assumes**  $jj < k$   
**and**  $tps = tpsL t xs 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (0, \text{Some } 0)$   
**and**  $tps' = tpsL t (xs[jj:= (1, \text{Some } 0)]) TT 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (tmL45 jj) tps esL45 tps'$   
 ⟨*proof*⟩

**definition**  $esL46 \equiv esL45 @ [(n + 1, \text{fmt } n)]$

**lemma** *len-esL46*:  $\text{length } esL46 = TT + 2$   
 ⟨*proof*⟩

**lemma** *tmL46*:  
**assumes**  $jj < k$   
**and**  $tps = tpsL t xs 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (0, \text{Some } 0)$   
**and**  $tps' = tpsL t (xs[jj:= (1, \text{Some } 0)]) (\text{fmt } n) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (tmL46 jj) tps esL46 tps'$   
 ⟨*proof*⟩

**lemma** *sem-cmdL7-nonleft-gt-0*:  
**assumes**  $jj < k$   
**and**  $tps = tpsL t xs i 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $i < TT$   
**and**  $i > 0$   
**and**  $\text{sim-move } t ! jj \neq 0$   
**and**  $tps' = tpsL t xs (i - 1) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{sem } (cmdL7 jj) (0, tps) = (0, tps')$   
 ⟨*proof*⟩

**lemma** *sem-cmdL7-nonleft-eq-0*:  
**assumes**  $jj < k$   
**and**  $tps = tpsL t xs 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $\text{sim-move } t ! jj \neq 0$   
**shows**  $\text{sem } (cmdL7 jj) (0, tps) = (1, tps)$   
 ⟨*proof*⟩

**lemma** *execute-tmL67-nonleft-less*:  
**assumes**  $jj < k$   
**and**  $tps = tpsL t xs (\text{fmt } n) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $\text{sim-move } t ! jj \neq 0$   
**and**  $tt < TT$   
**and**  $tps' = tpsL t xs (\text{fmt } n - tt) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{execute } (tmL67 jj) (0, tps) tt = (0, tps')$   
 ⟨*proof*⟩

**lemma** *execute-tmL67-nonleft-finish*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ (fmt\ n)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $sim-move\ t!\ jj \neq 0$   
**and**  $tps' = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $execute\ (tmL67\ jj)\ (0, tps)\ TT = (1, tps')$   
*<proof>*

**definition**  $esL67 \equiv map\ (\lambda i. (n + 1, i))\ (rev\ [0..<fmt\ n])\ @\ [(n + 1, 0)]$

**lemma**  $esL67-at-fmtn\ [simp]: esL67!\ (fmt\ n) = (n + 1, 0)$   
*<proof>*

**lemma**  $esL67-at-lt-fmtn\ [simp]: i < fmt\ n \implies esL67!\ i = (n + 1, fmt\ n - i - 1)$   
*<proof>*

**lemma**  $tmL67-nonleft:$   
**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ (fmt\ n)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $sim-move\ t!\ jj \neq 0$   
**and**  $tps' = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $traces\ (tmL67\ jj)\ tps\ esL67\ tps'$   
*<proof>*

**lemma**  $sem-cmdL7-1:$   
**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ i\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 0)$   
**and**  $i < TT$   
**and**  $i > exec\ t\ <\#\>\ jj$   
**and**  $sim-move\ t!\ jj = 0$   
**and**  $tps' = tpsL\ t\ xs\ (i - 1)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $sem\ (cmdL7\ jj)\ (0, tps) = (0, tps')$   
*<proof>*

**lemma**  $execute-tmL67-1:$   
**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ (fmt\ n)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 0)$   
**and**  $sim-move\ t!\ jj = 0$   
**and**  $tt < TT - exec\ t\ <\#\>\ jj$   
**and**  $tps' = tpsL\ t\ xs\ (fmt\ n - tt)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $execute\ (tmL67\ jj)\ (0, tps)\ tt = (0, tps')$   
*<proof>*

**lemma**  $zip-cont-enc-upd-Some:$   
**assumes**  $jj < k$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, None)$   
**and**  $i = exec\ (Suc\ t)\ <\#\>\ jj$   
**shows**  $(zip-cont\ t\ xs)(i := (enc-upd\ (zip-cont\ t\ xs\ i)\ (k + jj)\ 1)) = zip-cont\ t\ (xs[!jj := (1, Some\ 1)])$   
*(is ?lhs = ?rhs)*  
*<proof>*

**lemma**  $zip-cont-enc-upd-Some-Right:$   
**assumes**  $jj < k$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, None)$   
**and**  $i = Suc\ (exec\ t\ <\#\>\ jj)$   
**and**  $sim-move\ t!\ jj = 2$

**shows**  $(\text{zip-cont } t \text{ } xs)(i := (\text{enc-upd } (\text{zip-cont } t \text{ } xs \ i) \ (k + jj) \ 1)) = \text{zip-cont } t \ (xs[jj := (1, \text{Some } 1)])$   
 ⟨proof⟩

**lemma** *zip-cont-enc-upd-Some-Left*:

**assumes**  $jj < k$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{None})$   
**and**  $\text{Suc } i = \text{exec } t <\#\#> jj$   
**and**  $\text{sim-move } t ! jj = 0$   
**shows**  $(\text{zip-cont } t \text{ } xs)(i := (\text{enc-upd } (\text{zip-cont } t \text{ } xs \ i) \ (k + jj) \ 1)) = \text{zip-cont } t \ (xs[jj := (1, \text{Some } 1)])$   
 (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *zip-cont-enc-upd-None*:

**assumes**  $jj < k$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $i = \text{exec } t <\#\#> jj$   
**shows**  $(\text{zip-cont } t \text{ } xs)(i := (\text{enc-upd } (\text{zip-cont } t \text{ } xs \ i) \ (k + jj) \ 0)) = \text{zip-cont } t \ (xs[jj := (1, \text{None})])$   
 (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *sem-cmdL7-2a*:

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t \text{ } xs \ i \ 1 \ (\lambda j. \text{sim-move } t ! j) \ (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $i = \text{exec } t <\#\#> jj$   
**and**  $i > 0$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $tps' = \text{tpsL } t \ (xs[jj := (1, \text{None})]) \ (i - 1) \ 1 \ (\lambda j. \text{if } j = jj \text{ then } 3 \text{ else } \text{sim-move } t ! j) \ (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{sem } (\text{cmdL7 } jj) \ (0, tps) = (0, tps')$   
 ⟨proof⟩

**lemma** *execute-tmL67-2a*:

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t \text{ } xs \ (\text{fmt } n) \ 1 \ (\lambda j. \text{sim-move } t ! j) \ (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $\text{exec } t <\#\#> jj > 0$   
**and**  $tt = TT - \text{exec } t <\#\#> jj$   
**and**  $tps' = \text{tpsL } t \ (xs[jj := (1, \text{None})]) \ (\text{fmt } n - tt) \ 1 \ (\lambda j. \text{if } j = jj \text{ then } 3 \text{ else } \text{sim-move } t ! j) \ (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{execute } (\text{tmL67 } jj) \ (0, tps) \ tt = (0, tps')$   
 ⟨proof⟩

**lemma** *zip-cont-Some-Some*:

**assumes**  $jj < k$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $i = \text{exec } t <\#\#> jj$   
**and**  $i = 0$   
**and**  $\text{sim-move } t ! jj = 0$   
**shows**  $\text{zip-cont } t \text{ } xs = \text{zip-cont } t \ (xs[jj := (1, \text{Some } 1)])$   
 (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *sem-cmdL7-2b*:

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t \text{ } xs \ i \ 1 \ (\lambda j. \text{sim-move } t ! j) \ (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$

**and**  $i = \text{exec } t <\#\> jj$   
**and**  $i = 0$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $\text{tps}' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) i 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{sem } (\text{cmdL7 } jj) (0, \text{tps}) = (1, \text{tps}'^{\wedge})$   
 $\langle \text{proof} \rangle$

**lemma** *execute-tmL67-2b*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t xs (\text{fmt } n) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $\text{exec } t <\#\> jj = 0$   
**and**  $\text{tps}' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{execute } (\text{tmL67 } jj) (0, \text{tps}) TT = (1, \text{tps}'^{\wedge})$   
 $\langle \text{proof} \rangle$

**lemma** *tmL67-left-0*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t xs (\text{fmt } n) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $\text{exec } t <\#\> jj = 0$   
**and**  $\text{tps}' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (\text{tmL67 } jj) \text{tps } \text{esL67 } \text{tps}'$   
 $\langle \text{proof} \rangle$

**lemma** *sem-cmdL7-3*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t xs i 1 (\lambda j. \text{if } j = jj \text{ then } \exists \text{ else } \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{None})$   
**and**  $\text{Suc } i = \text{exec } t <\#\> jj$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $\text{tps}' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) (i - 1) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{sem } (\text{cmdL7 } jj) (0, \text{tps}) = (\text{if } i = 0 \text{ then } 1 \text{ else } 0, \text{tps}'^{\wedge})$   
 $\langle \text{proof} \rangle$

**lemma** *execute-tmL67-3*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t xs (\text{fmt } n) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $\text{exec } t <\#\> jj > 0$   
**and**  $tt = TT - \text{exec } t <\#\> jj$   
**and**  $\text{tps}' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) (\text{fmt } n - \text{Suc } tt) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{execute } (\text{tmL67 } jj) (0, \text{tps}) (\text{Suc } tt) = (\text{if } \text{fmt } n - tt = 0 \text{ then } 1 \text{ else } 0, \text{tps}'^{\wedge})$   
 $\langle \text{proof} \rangle$

**lemma** *sem-cmdL7-4*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t xs i 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 1)$   
**and**  $\text{Suc } i < \text{exec } t <\#\> jj$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $\text{tps}' = \text{tpsL } t xs (i - 1) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{sem } (\text{cmdL7 } jj) (0, \text{tps}) = (\text{if } i = 0 \text{ then } 1 \text{ else } 0, \text{tps}'^{\wedge})$   
 $\langle \text{proof} \rangle$

**lemma** *execute-tmL67-4*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ (fmt\ n)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 0)$   
**and**  $sim-move\ t!\ jj = 0$   
**and**  $exec\ t <\#\>\ jj > 0$   
**and**  $tt \geq Suc\ (Suc\ (TT - exec\ t <\#\>\ jj))$   
**and**  $tt \leq TT$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 1)])\ (fmt\ n - tt)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $execute\ (tmL67\ jj)\ (0, tps)\ tt = (if\ TT - tt = 0\ then\ 1\ else\ 0, tps')$   
*<proof>*

**lemma** *tmL67-left-gt-0*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ (fmt\ n)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 0)$   
**and**  $sim-move\ t!\ jj = 0$   
**and**  $exec\ t <\#\>\ jj > 0$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 1)])\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $traces\ (tmL67\ jj)\ tps\ esL67\ tps'$   
*<proof>*

**lemma** *tmL67-left*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ (fmt\ n)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 0)$   
**and**  $sim-move\ t!\ jj = 0$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 1)])\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $traces\ (tmL67\ jj)\ tps\ esL67\ tps'$   
*<proof>*

**definition**  $esL47 \equiv esL46\ @\ esL67$

**lemma** *len-esL47*:  $length\ esL47 = 2 * TT + 2$

*<proof>*

**lemma** *tmL47-nonleft*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (0, Some\ 0)$   
**and**  $sim-move\ t!\ jj \neq 0$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 0)])\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $traces\ (tmL47\ jj)\ tps\ esL47\ tps'$   
*<proof>*

**lemma** *tmL47-left*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (0, Some\ 0)$   
**and**  $sim-move\ t!\ jj = 0$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 1)])\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $traces\ (tmL47\ jj)\ tps\ esL47\ tps'$   
*<proof>*

**lemma** *sem-cmdL8-nonright*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ i\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$

**and**  $i < TT$   
**and**  $\text{sim-move } t ! jj \neq 2$   
**and**  $\text{tps}' = \text{tpsL } t \text{ xs } (\text{Suc } i) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{sem } (\text{cmdL8 } jj) (0, \text{tps}) = (0, \text{tps}')$   
 ⟨proof⟩

**lemma** *sem-cmdL8-TT*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t \text{ xs } i 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } \text{xs} = k$   
**and**  $i = TT$   
**shows**  $\text{sem } (\text{cmdL8 } jj) (0, \text{tps}) = (1, \text{tps})$   
 ⟨proof⟩

**lemma** *execute-tmL78-nonright-le-TT*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t \text{ xs } 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } \text{xs} = k$   
**and**  $\text{sim-move } t ! jj \neq 2$   
**and**  $tt \leq TT$   
**and**  $\text{tps}' = \text{tpsL } t \text{ xs } tt 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{execute } (\text{tmL78 } jj) (0, \text{tps}) tt = (0, \text{tps}')$   
 ⟨proof⟩

**lemma** *execute-tmL78-nonright-eq-Suc-TT*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t \text{ xs } 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } \text{xs} = k$   
**and**  $\text{sim-move } t ! jj \neq 2$   
**and**  $\text{tps}' = \text{tpsL } t \text{ xs } TT 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{execute } (\text{tmL78 } jj) (0, \text{tps}) (\text{Suc } TT) = (1, \text{tps}')$   
 ⟨proof⟩

**definition**  $\text{esL78} \equiv \text{map } (\lambda i. (n + 1, \text{Suc } i)) ([0..<TT]) @ [(n + 1, TT)]$

**lemma** *tmL78-nonright*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t \text{ xs } 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } \text{xs} = k$   
**and**  $\text{sim-move } t ! jj \neq 2$   
**and**  $\text{tps}' = \text{tpsL } t \text{ xs } TT 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (\text{tmL78 } jj) \text{tps esL78 } \text{tps}'$   
 ⟨proof⟩

**lemma** *sem-cmdL8-1*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t \text{ xs } i 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } \text{xs} = k$   
**and**  $\text{xs} ! jj = (1, \text{Some } 0)$   
**and**  $i < \text{exec } t <\#\#> jj$   
**and**  $\text{sim-move } t ! jj = 2$   
**and**  $\text{tps}' = \text{tpsL } t \text{ xs } (\text{Suc } i) 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{sem } (\text{cmdL8 } jj) (0, \text{tps}) = (0, \text{tps}')$   
 ⟨proof⟩

**lemma** *execute-tmL78-1*:

**assumes**  $jj < k$   
**and**  $\text{tps} = \text{tpsL } t \text{ xs } 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } \text{xs} = k$   
**and**  $\text{xs} ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 2$   
**and**  $tt \leq \text{exec } t <\#\#> jj$   
**and**  $\text{tps}' = \text{tpsL } t \text{ xs } tt 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$

**shows**  $execute (tmL78\ jj) (0, tps) tt = (0, tps')$   
 ⟨proof⟩

**lemma** *sem-cmdL8-2*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ i\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 0)$   
**and**  $i = exec\ t\ <\#\>\ jj$   
**and**  $sim-move\ t!\ jj = 2$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, None)])\ (Suc\ i)\ 1\ (\lambda j. if\ j = jj\ then\ 3\ else\ sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $sem\ (cmdL8\ jj) (0, tps) = (0, tps')$   
 ⟨proof⟩

**lemma** *execute-tmL78-2*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 0)$   
**and**  $sim-move\ t!\ jj = 2$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, None)])\ (Suc\ (exec\ t\ <\#\>\ jj))\ 1\ (\lambda j. if\ j = jj\ then\ 3\ else\ sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $execute (tmL78\ jj) (0, tps) (Suc\ (exec\ t\ <\#\>\ jj)) = (0, tps')$   
 ⟨proof⟩

**lemma** *sem-cmdL8-3*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ i\ 1\ (\lambda j. if\ j = jj\ then\ 3\ else\ sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, None)$   
**and**  $i = Suc\ (exec\ t\ <\#\>\ jj)$   
**and**  $sim-move\ t!\ jj = 2$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 1)])\ (Suc\ i)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $sem\ (cmdL8\ jj) (0, tps) = (0, tps')$   
 ⟨proof⟩

**lemma** *execute-tmL78-3*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 0)$   
**and**  $sim-move\ t!\ jj = 2$   
**and**  $tps' = tpsL\ t\ (xs[jj:= (1, Some\ 1)])\ (Suc\ (Suc\ (exec\ t\ <\#\>\ jj)))\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $execute (tmL78\ jj) (0, tps) (Suc\ (Suc\ (exec\ t\ <\#\>\ jj))) = (0, tps')$   
 ⟨proof⟩

**lemma** *sem-cmdL8-4*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ i\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$   
**and**  $xs!\ jj = (1, Some\ 1)$   
**and**  $i > Suc\ (exec\ t\ <\#\>\ jj)$   
**and**  $i < TT$   
**and**  $sim-move\ t!\ jj = 2$   
**and**  $tps' = tpsL\ t\ xs\ (Suc\ i)\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $sem\ (cmdL8\ jj) (0, tps) = (0, tps')$   
 ⟨proof⟩

**lemma** *execute-tmL78-4*:

**assumes**  $jj < k$   
**and**  $tps = tpsL\ t\ xs\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**and**  $length\ xs = k$

**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 2$   
**and**  $tt \geq \text{Suc } (\text{Suc } (\text{exec } t <\#\> jj))$   
**and**  $tt \leq TT$   
**and**  $tps' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) tt 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{execute } (\text{tmL78 } jj) (0, tps) tt = (0, tps')$   
*<proof>*

**lemma** *execute-tmL78-5:*

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t xs 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 2$   
**and**  $tps' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) TT 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{execute } (\text{tmL78 } jj) (0, tps) (\text{Suc } TT) = (1, tps')$   
*<proof>*

**lemma** *tmL78-right:*

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t xs 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 2$   
**and**  $tps' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) TT 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (\text{tmL78 } jj) tps \text{ esL78 } tps'$   
*<proof>*

**lemma** *zip-cont-Stay:*

**assumes**  $jj < k$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 1$   
**shows**  $\text{zip-cont } t xs = \text{zip-cont } t (xs[jj:= (1, \text{Some } 1)])$   
*<proof>*

**lemma** *tpsL-Stay:*

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t xs i 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (1, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 1$   
**shows**  $tps = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) i 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
*<proof>*

**definition**  $\text{esL48} \equiv \text{esL47} @ \text{esL78}$

**lemma** *len-esL48:*  $\text{length } \text{esL48} = 3 * TT + 3$

*<proof>*

**lemma** *tmL48-left:*

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t xs 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (0, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 0$   
**and**  $tps' = \text{tpsL } t (xs[jj:= (1, \text{Some } 1)]) TT 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (\text{tmL48 } jj) tps \text{ esL48 } tps'$   
*<proof>*

**lemma** *tmL48-right:*

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t xs 0 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$

**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (0, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 2$   
**and**  $tps' = \text{tpsL } t (xs[jj := (1, \text{Some } 1)]) \text{ TT } 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (\text{tmL48 } jj) \text{ tps } \text{esL48 } tps'$   
 <proof>

**lemma** *tmL48-stay*:

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t \text{ xs } 0 \ 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (0, \text{Some } 0)$   
**and**  $\text{sim-move } t ! jj = 1$   
**and**  $tps' = \text{tpsL } t (xs[jj := (1, \text{Some } 1)]) \text{ TT } 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (\text{tmL48 } jj) \text{ tps } \text{esL48 } tps'$   
 <proof>

**lemma** *tmL48*:

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t \text{ xs } 0 \ 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (0, \text{Some } 0)$   
**and**  $tps' = \text{tpsL } t (xs[jj := (1, \text{Some } 1)]) \text{ TT } 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (\text{tmL48 } jj) \text{ tps } \text{esL48 } tps'$   
 <proof>

**definition**  $\text{esL49} \equiv \text{esL48} @ \text{map } (\lambda i. (n + 1, i)) (\text{rev } [0..< \text{TT}]) @ [(n + 1, 0)]$

**lemma** *len-esL49*:  $\text{length } \text{esL49} = 4 * \text{TT} + 4$

<proof>

**lemma** *tmL49*:

**assumes**  $jj < k$   
**and**  $tps = \text{tpsL } t \text{ xs } 0 \ 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**and**  $\text{length } xs = k$   
**and**  $xs ! jj = (0, \text{Some } 0)$   
**and**  $tps' = \text{tpsL } t (xs[jj := (1, \text{Some } 1)]) \ 0 \ 1 (\lambda j. \text{sim-move } t ! j) (\lambda j. \text{sim-write } t ! j)$   
**shows**  $\text{traces } (\text{tmL49 } jj) \text{ tps } \text{esL49 } tps'$   
 <proof>

**definition**  $xs49 :: \text{nat} \Rightarrow (\text{nat} \times \text{nat option}) \text{ list}$  **where**

$xs49 \ j \equiv \text{replicate } j \ (1, \text{Some } 1) @ \text{replicate } (k - j) \ (0, \text{Some } 0)$

**lemma** *length-xs49*:  $j \leq k \implies \text{length } (xs49 \ j) = k$

<proof>

**lemma** *xs49-less*:

**assumes**  $j \leq k$  **and**  $i < j$   
**shows**  $xs49 \ j ! i = (1, \text{Some } 1)$   
 <proof>

**lemma** *xs49-ge*:

**assumes**  $j \leq k$  **and**  $i \geq j$  **and**  $i < k$   
**shows**  $xs49 \ j ! i = (0, \text{Some } 0)$   
 <proof>

**lemma** *xs49-upd*:

**assumes**  $j < k$   
**shows**  $xs49 \ (\text{Suc } j) = (xs49 \ j)[j := (1, \text{Some } 1)]$   
 (is ?lhs = ?rhs)  
 <proof>

**lemma** *tmL49-upt*:

**assumes**  $j \leq k$   
**and**  $tps' = tpsL\ t\ (xs49\ j)\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $traces\ (tmL49-upt\ j)\ (tpsL4\ t)\ (concat\ (replicate\ j\ esL49))\ tps'$   
 $\langle proof \rangle$

**definition**  $esL49-upt \equiv concat\ (replicate\ k\ esL49)$

**lemma**  $length-concat-replicate: length\ (concat\ (replicate\ m\ xs)) = m * length\ xs$   
 $\langle proof \rangle$

**lemma**  $len-esL49-upt: length\ esL49-upt = k * (4 * TT + 4)$   
 $\langle proof \rangle$

**corollary**  $tmL49-upt'$ :  
**assumes**  $tps' = tpsL\ t\ (xs49\ k)\ 0\ 1\ (\lambda j. sim-move\ t!\ j)\ (\lambda j. sim-write\ t!\ j)$   
**shows**  $traces\ (tmL49-upt\ k)\ (tpsL4\ t)\ esL49-upt\ tps'$   
 $\langle proof \rangle$

**definition**  $esL9\ t \equiv esL4\ t\ @\ esL49-upt$

**lemma**  $len-esL9: length\ (esL9\ t) = k * (4 * TT + 4) + t + 2 * TT + 4$   
 $\langle proof \rangle$

**lemma**  $xs49-k: xs49\ k = replicate\ k\ (1, Some\ 1)$   
 $\langle proof \rangle$

**definition**  $tpsL9\ t \equiv tpsL$   
 $t$   
 $(replicate\ k\ (1, Some\ 1))$   
 $0$   
 $1$   
 $(\lambda j. sim-move\ t!\ j)$   
 $(\lambda j. sim-write\ t!\ j)$

**lemma**  $tmL9$ :  
**assumes**  $t < TT$   
**shows**  $traces\ tmL9\ (tpsL0\ t)\ (esL9\ t)\ (tpsL9\ t)$   
 $\langle proof \rangle$

**definition**  $esL10\ t \equiv esL9\ t\ @\ esC\ t$

**lemma**  $len-esL10: length\ (esL10\ t) = k * (4 * TT + 4) + 2 * t + 2 * TT + 5$   
 $\langle proof \rangle$

**definition**  $tpsL10\ t \equiv tpsL$   
 $t$   
 $(replicate\ k\ (1, Some\ 1))$   
 $t$   
 $1$   
 $(\lambda j. sim-move\ t!\ j)$   
 $(\lambda j. sim-write\ t!\ j)$

**lemma**  $tmL10$ :  
**assumes**  $t < TT$   
**shows**  $traces\ tmL10\ (tpsL0\ t)\ (esL10\ t)\ (tpsL10\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL11\ t \equiv tpsL$   
 $(Suc\ t)$   
 $(replicate\ k\ (0, Some\ 0))$   
 $t$   
 $0$   
 $(\lambda j. sim-move\ t!\ j)$

( $\lambda j. \text{sim-write } t ! j$ )

**lemma** *enc-upd-2k*:

**assumes**  $\text{dec } n = (\text{map } f [0..<k] @ \text{map } h [0..<k] @ [a, b])$   
**shows**  $\text{enc-upd } n (2 * k) 1 = \text{enc } (\text{map } f [0..<k] @ \text{map } h [0..<k] @ [1, b])$   
(*proof*)

**lemma** *enc-upd-zip-cont*:

**assumes**  $t < TT$   
**and**  $xs1 = \text{replicate } k (1, \text{Some } 1)$   
**and**  $xs0 = (\text{replicate } k (0, \text{Some } 0))$   
**shows**  $\text{enc-upd } (\text{zip-cont } t xs1 t) (2 * k) 1 = \text{zip-cont } (\text{Suc } t) xs0 t$   
(*proof*)

**lemma** *enc-upd-zip-cont-upd*:

**assumes**  $t < TT$   
**and**  $xs1 = \text{replicate } k (1, \text{Some } 1)$   
**and**  $xs0 = (\text{replicate } k (0, \text{Some } 0))$   
**shows**  $(\text{zip-cont } t xs1) (t := \text{enc-upd } (\text{zip-cont } t xs1 t) (2 * k) 1) = \text{zip-cont } (\text{Suc } t) xs0$   
(*proof*)

**lemma** *sem-cmdL11*:

**assumes**  $t < TT$   
**shows**  $\text{sem cmdL11 } (0, \text{tpsL10 } t) = (1, \text{tpsL11 } t)$   
(*proof*)

**definition**  $esL11 t \equiv esL10 t @ [(n + 1, t)]$

**lemma** *len-esL11*:  $\text{length } (esL11 t) = k * (4 * TT + 4) + 2 * t + 2 * TT + 6$   
(*proof*)

**lemma** *tmL11*:

**assumes**  $t < TT$   
**shows**  $\text{traces } tmL11 (\text{tpsL0 } t) (esL11 t) (\text{tpsL11 } t)$   
(*proof*)

**definition**  $esL12 t \equiv esL11 t @ \text{map } (\lambda i. (n + 1, i)) (\text{rev } [0..<t]) @ [(n + 1, 0)]$

**lemma** *len-esL12*:  $\text{length } (esL12 t) = k * (4 * TT + 4) + 3 * t + 2 * TT + 7$   
(*proof*)

**definition**  $tpsL12 t \equiv \text{tpsL}$

(*Suc t*)  
(*replicate k (0, Some 0)*)  
*0*  
*0*  
( $\lambda j. \text{sim-move } t ! j$ )  
( $\lambda j. \text{sim-write } t ! j$ )

**lemma** *tmL12*:

**assumes**  $t < TT$   
**shows**  $\text{traces } tmL12 (\text{tpsL0 } t) (esL12 t) (\text{tpsL12 } t)$   
(*proof*)

**definition**  $tpsL13 t \equiv \text{tpsL}$

(*Suc t*)  
(*replicate k (0, Some 0)*)  
*0*  
*0*  
( $\lambda j. 0$ )  
( $\lambda j. 0$ )

**definition**  $esL13 t \equiv esL12 t @ [(n + 1, 0)]$

**lemma** *len-esL13*:  $\text{length } (esL13\ t) = k * (4 * TT + 4) + 3 * t + 2 * TT + 8$   
 ⟨proof⟩

**lemma** *tmL13*:  
**assumes**  $t < TT$   
**shows** *traces tmL13* (tpsL0 t) (esL13 t) (tpsL13 t)  
 ⟨proof⟩

**corollary** *tmL13'*:  
**assumes**  $t < TT$   
**shows** *traces tmL13* (tpsC1 t) (esL13 t) (tpsL13 t)  
 ⟨proof⟩

**definition** *esLoop-while*  $t \equiv$   
 $esC\ t\ @\ [(tpsC1\ t\ :\#: 0, tpsC1\ t\ :\#: 1)]\ @\ esL13\ t\ @\ [(tpsL13\ t\ :\#: 0, tpsL13\ t\ :\#: 1)]$

**definition** *esLoop-break*  $\equiv (esC\ TT)\ @\ [(tpsC1\ TT\ :\#: 0, tpsC1\ TT\ :\#: 1)]$

**lemma** *len-esLoop-while*:  $\text{length } (esLoop\text{-}while\ t) = k * (4 * TT + 4) + 4 * t + 2 * TT + 11$   
 ⟨proof⟩

**lemma** *tmLoop-while*:  
**assumes**  $t < TT$   
**shows** *trace tmLoop* (0, tpsC0 t) (esLoop-while t) (0, tpsL13 t)  
 ⟨proof⟩

**lemma** *tmLoop-while-end*:  
*trace tmLoop* (0, tpsC0 0) (concat (map esLoop-while [0.. $TT$ ])) (0, tpsC0  $TT$ )  
 ⟨proof⟩

**lemma** *len-esLoop-break*:  $\text{length } esLoop\text{-}break = TT + 2$   
 ⟨proof⟩

**lemma** *tmLoop-break*: *traces tmLoop* (tpsC0  $TT$ ) esLoop-break (tpsC1  $TT$ )  
 ⟨proof⟩

**definition** *esLoop*  $\equiv$  concat (map esLoop-while [0.. $TT$ ]) @ esLoop-break

**lemma** *len-esLoop1*:  $u \leq TT \implies \text{length } (concat\ (map\ esLoop\text{-}while\ [0..\langle u \rangle])) \leq u * (k * (4 * TT + 4) + 4 * TT + 2 * TT + 11)$   
 ⟨proof⟩

**lemma** *len-esLoop2*:  $\text{length } (concat\ (map\ esLoop\text{-}while\ [0..\langle TT \rangle])) \leq TT * (k * (4 * TT + 4) + 4 * TT + 2 * TT + 11)$   
 ⟨proof⟩

**lemma** *len-esLoop3*:  $\text{length } esLoop \leq TT * (k * (4 * TT + 4) + 4 * TT + 2 * TT + 11) + TT + 2$   
 ⟨proof⟩

**lemma** *len-esLoop*:  $\text{length } esLoop \leq 28 * k * TT * TT$   
 ⟨proof⟩

**lemma** *tmLoop*: *traces tmLoop* (tpsC0 0) esLoop (tpsC1  $TT$ )  
 ⟨proof⟩

**lemma** *tps9-tpsC0*:  $tps9 = tpsC0\ 0$   
 ⟨proof⟩

**definition** *es10*  $\equiv es9\ @\ esLoop$

**lemma** *len-es10*:  $\text{length } es10 \leq \text{length } (es\text{-}fmt\ n) + 40 * k * TT * TT$   
 ⟨proof⟩

**lemma** *tm10*: traces *tm10 tps0 es10 (tpsC1 TT)*  
 ⟨proof⟩

### Cleaning up the output

**abbreviation** *tps10*  $\equiv$  *tpsC1 TT*

**definition** *es11*  $\equiv$  *es10 @ map* ( $\lambda i. (n + 1, i)$ ) (*rev* [ $0..<TT$ ]) @ [( $n + 1, 0$ )]

**lemma** *len-es11*: *length es11*  $\leq$  *length (es-fmt n)* +  $40 * k * TT * TT + Suc TT$   
 ⟨proof⟩

**definition** *tps11*  $\equiv$  *tps10*[ $1 := ltransplant (tps10 ! 1) (tps10 ! 1) ec1 TT$ ]

**lemma** *tm11*: traces *tm11 tps0 es11 tps11*  
 ⟨proof⟩

**definition** *es12*  $\equiv$  *es11 @ [(n + 1, 1)]*

The upper bound on the length of the trace will help us establish an upper bound of the total running time.

**lemma** *length-es12*: *length es12*  $\leq$  *length (es-fmt n)* +  $43 * k * TT * TT$   
 ⟨proof⟩

**definition** *tps12*  $\equiv$  *tps11*[ $1 := tps11 ! 1 |:=| (ec1 (tps11 :: 1)) | + | 1$ ]

**lemma** *tm12*: traces *tm12 tps0 es12 tps12*  
 ⟨proof⟩

**lemma** *tps11-0*: (*tps11 :: 1*) 0 = (*zip-cont TT (replicate k (0, Some 0))*) 0  
 ⟨proof⟩

**lemma** *tps11-gr0-exec*:  
**assumes**  $i > 0$   
**shows** (*tps11 :: 1*)  $i = (exec TT <:> 1) i$   
 ⟨proof⟩

**definition** *tps12'*  $\equiv$   
 [(*zs*],  $n + 1$ ),  
 (*exec TT <:> 1, 1*),  
 [*fst (exec TT)*]] @  
*map* ( $\lambda i. [\square]$ ) [ $0..<k$ ]  
*map* ( $\lambda i. [\square]$ ) [ $0..<k$ ]

**lemma** *tps12'*: *tps12'* = *tps12*  
 ⟨proof⟩

**lemma** *tm12'*: traces *tm12 tps0 es12 tps12'*  
 ⟨proof⟩

**end**

### 5.3.4 Shrinking the Turing machine to two tapes

The simulator TM *tm12* has  $2k + 3$  tapes, of which  $2k + 1$  are immobile and thus can be removed by the memorization-in-states technique, resulting in a two-tape TM.

**lemma** *immobile-tm12*:  
**assumes**  $j > 1$  **and**  $j < 2 * k + 3$   
**shows** *immobile tm12 j* ( $2 * k + 3$ )  
 ⟨proof⟩

**definition** *tps12'' zs*  $\equiv$

$[(\lfloor zs \rfloor, \text{length } zs + 1),$   
 $(\text{exec } zs \text{ (Suc (fmt (length } zs)))} \langle \cdot \rangle 1, 1)]$

**lemma** *tps12''*:  
**assumes** *bit-symbols* *zs*  
**shows**  $\text{tps12'' } zs = \text{take } 2 \text{ (tps12' } zs)$   
 $\langle \text{proof} \rangle$

This is the actual simulator Turing machine we are constructing in this section. It is *tm12* stripped of all memorization tapes:

**definition**  $\text{tmO2T} \equiv \text{icartesian } (2 * k + 1) \text{ tm12 } G'$

**lemma** *tmO2T-tm: turing-machine*  $2 \text{ } G' \text{ tmO2T}$   
 $\langle \text{proof} \rangle$

The constructed two-tape Turing machine computes the same output as the original Turing machine.

**lemma** *tmO2T*:  
**assumes** *bit-symbols* *zs*  
**shows**  $\text{traces } \text{tmO2T} \text{ (snd (start-config } 2 \text{ } zs)) \text{ (es12 } zs) \text{ (tps12'' } zs)$   
 $\langle \text{proof} \rangle$

### 5.3.5 Time complexity

This is the bound for the running time of *tmO2T*:

**definition**  $\text{TTT} :: \text{nat} \Rightarrow \text{nat}$  **where**  
 $\text{TTT} \equiv \lambda n. \text{length (es-fmt } n) + 43 * k * \text{Suc (fmt } n) * \text{Suc (fmt } n)$

**lemma** *execute-tmO2T*:  
**assumes** *bit-symbols* *zs*  
**shows**  $\text{execute } \text{tmO2T} \text{ (start-config } 2 \text{ } zs) \text{ (TTT (length } zs)) = (\text{length } \text{tmO2T}, \text{tps12'' } zs)$   
 $\langle \text{proof} \rangle$

The simulator TM *tmO2T* halts with the output tape head on cell 1.

**lemma** *execute-tmO2T-1*:  
**assumes** *bit-symbols* *zs*  
**shows**  $\text{execute } \text{tmO2T} \text{ (start-config } 2 \text{ } zs) \text{ (TTT (length } zs)) \langle \cdot \rangle 1 =$   
 $(\text{execute } M \text{ (start-config } k \text{ } zs) \text{ (T (length } zs))} \langle \cdot \rangle 1, 1)$   
 $\langle \text{proof} \rangle$

**lemma** *poly-TTT: big-oh-poly*  $\text{TTT}$   
 $\langle \text{proof} \rangle$

### 5.3.6 Obliviousness

The two-tape simulator machine is oblivious.

**lemma** *tmO2T-oblivious*:  
**assumes**  $\text{length } zs1 = \text{length } zs2$  **and** *bit-symbols* *zs1* **and** *bit-symbols* *zs2*  
**shows**  $\text{es12 } zs1 = \text{es12 } zs2$   
 $\langle \text{proof} \rangle$

**end**

## 5.4 $\mathcal{NP}$ and obliviousness

This section presents the main result of this chapter: For every language  $L \in \mathcal{NP}$  there is a polynomial-time two-tape oblivious verifier TM that halts with the output tape head on a **1** symbol iff. in the input  $\langle x, u \rangle$ , the string  $u$  is a certificate for  $x$ . The proof combines two lemmas. First *NP-output-len-1*, which says that we can assume the verifier outputs only one symbol (namely, **0** or **1**), and second *two-tape.execute-tmO2T-1*, which says that the two-tape oblivious TM halts with output tape head in cell 1. This cell will contain either **0** or **1** by the first lemma.

**lemma** *NP-imp-oblivious-2tape:*

**fixes**  $L :: \text{language}$

**assumes**  $L \in \mathcal{NP}$

**obtains**  $M G T p$  **where**

*big-oh-poly*  $T$  **and**

*polynomial*  $p$  **and**

*turing-machine*  $2 G M$  **and**

*oblivious*  $M$  **and**

$\bigwedge y. \text{bit-symbols } y \implies \text{fst } (\text{execute } M \text{ (start-config } 2 y) (T (\text{length } y))) = \text{length } M$  **and**

$\bigwedge x. x \in L \longleftrightarrow (\exists u. \text{length } u = p (\text{length } x) \wedge \text{execute } M \text{ (start-config } 2 \langle x; u \rangle) (T (\text{length } \langle x; u \rangle)) \langle \cdot \rangle 1$

$= \mathbf{1})$

*<proof>*

**end**

# Chapter 6

## Reducing $\mathcal{NP}$ languages to SAT

**theory** *Reducing*  
**imports** *Satisfiability Oblivious*  
**begin**

We have already shown that SAT is in  $\mathcal{NP}$ . It remains to show that SAT is  $\mathcal{NP}$ -hard, that is, that every language  $L \in \mathcal{NP}$  can be polynomial-time reduced to SAT. This, in turn, can be split in two parts. First, showing that for every  $x$  there is a CNF formula  $\Phi$  such that  $x \in L$  iff.  $\Phi$  is satisfiable. Second, that  $\Phi$  can be computed from  $x$  in polynomial time. This chapter is devoted to the first part, which is the core of the proof. In the subsequent two chapters we painstakingly construct a polynomial-time Turing machine computing  $\Phi$  from  $x$  in order to show something that is usually considered “obvious”.

The proof corresponds to lemma 2.11 from the textbook [2]. Of course we have to be much more explicit than the textbook, and the first section describes in some detail how we derive the formula  $\Phi$ .

### 6.1 Introduction

Let  $L \in \mathcal{NP}$ . In order to reduce  $L$  to SAT, we need to construct for every string  $x \in \{\mathbf{0}, \mathbf{1}\}^*$  a CNF formula  $\Phi$  such that  $x \in L$  iff.  $\Phi$  is satisfiable. In this section we describe how  $\Phi$  looks like.

#### 6.1.1 Preliminaries

We denote the length of a string  $s \in \{\mathbf{0}, \mathbf{1}\}^*$  by  $|s|$ . We define

$$\text{num}(s) = \begin{cases} k & \text{if } s = \mathbf{1}^k \mathbf{0}^{|s|-k}, \\ |s| + 1 & \text{otherwise.} \end{cases}$$

Essentially  $\text{num}$  interprets some strings as unary codes of numbers. All other strings are interpreted as an “error value”.

For a string  $s$  and a sequence  $w \in \mathbb{N}^n$  of numbers we write  $s(w)$  for  $\text{num}(s_{w_0} \dots s_{w_{n-1}})$ . Likewise for an assignment  $\alpha: \mathbb{N} \rightarrow \{\mathbf{0}, \mathbf{1}\}$  we write  $\alpha(w) = \text{num}(\alpha(w_0) \dots \alpha(w_{n-1}))$ .

We define two families of CNF formulas. Variables are written  $v_0, v_1, v_2, \dots$ , and negated variables are written  $\bar{v}_0, \bar{v}_1, \bar{v}_2, \dots$ . Let  $w \in \mathbb{N}^n$  be a list of numbers. For  $k \leq n$  define

$$\Psi(w, k) = \bigwedge_{i=0}^{k-1} v_{w_i} \wedge \bigwedge_{i=k}^{n-1} \bar{v}_{w_i}.$$

This formula is satisfied by an assignment  $\alpha$  iff.  $\alpha(w) = k$ . In a similar fashion we define for  $n > 2$ ,

$$\Upsilon(w) = v_{w_0} \wedge v_{w_1} \wedge \bigwedge_{i=3}^{n-1} \bar{v}_{w_i},$$

which is satisfied by an assignment  $\alpha$  iff.  $\alpha(w) \in \{2, 3\} = \{\mathbf{0}, \mathbf{1}\}$ , where as usual the boldface  $\mathbf{0}$  and  $\mathbf{1}$  refer to the symbols represented by the numbers 2 and 3.

For  $a \leq b$  we write  $[a : b]$  for the interval  $[a, \dots, b - 1] \in \mathbb{N}^{b-a}$ . For intervals the CNF formulas become:

$$\Psi([a : b], k) = \bigwedge_{i=a}^{a+k-1} v_i \wedge \bigwedge_{i=a+k}^{b-1} \bar{v}_i \quad \text{and} \quad \Upsilon([a : b]) = v_a \wedge v_{a+1} \wedge \bigwedge_{i=a+3}^{b-1} \bar{v}_i.$$

Let  $\varphi$  be a CNF formula and let  $\sigma \in \mathbb{N}^*$  be a sequence of variable indices such that for all variables  $v_i$  occurring in  $\varphi$  we have  $i < |\sigma|$ . Then we define the CNF formula  $\sigma(\varphi)$  as the formula resulting from replacing every variable  $v_i$  in  $\varphi$  by the variable  $v_{\sigma_i}$ . This corresponds to our function *relabel*.

### 6.1.2 Construction of the CNF formula

Let  $M$  be the two-tape oblivious verifier Turing machine for  $L$  from lemma *NP-imp-oblivious-2tape*. Let  $p$  be the polynomial function for the length of the certificates, and let  $T: \mathbb{N} \rightarrow \mathbb{N}$  be the polynomial running-time bound. Let  $G$  be  $M$ 's alphabet size.

Let  $x \in \{\mathbf{0}, \mathbf{1}\}^n$  be fixed throughout the rest of this section. We seek a CNF formula  $\Phi$  that is satisfiable iff.  $x \in L$ . We are going to transform “ $x \in L$ ” via several equivalent statements to the statement “ $\Phi$  is satisfiable” for a suitable  $\Phi$  defined along the way. The Isabelle formalization later in this chapter does not prove these equivalences explicitly. They are only meant to explain the shape of  $\Phi$ .

#### 1st equivalence

From lemma *NP-imp-oblivious-2tape* about  $M$  we get the first equivalent statement: There exists a certificate  $u \in \{\mathbf{0}, \mathbf{1}\}^{p(n)}$  such that  $M$  on input  $\langle x, u \rangle$  halts with the symbol  $\mathbf{1}$  under its output tape head. The running time of  $M$  is bounded by  $T(|\langle x, u \rangle|) = T(2n + 2 + 2p(n))$ . We abbreviate  $|\langle x, u \rangle| = 2n + 2 + 2p(n)$  by  $m$ .

#### 2nd equivalence

For the second equivalent statement, we define what the textbook calls “snapshots”. For every  $u \in \{\mathbf{0}, \mathbf{1}\}^{p(n)}$  let  $z_0^u(t)$  be the symbol under the input tape head of  $M$  on input  $\langle x, u \rangle$  at step  $t$ . Similarly we define  $z_1^u(t)$  as the symbol under the output tape head of  $M$  at step  $t$  and  $z_2^u(t)$  as the state  $M$  is in at step  $t$ . A triple  $z^u(t) = (z_0^u(t), z_1^u(t), z_2^u(t))$  is called a snapshot. For the initial snapshot we have:

$$z_0^u(0) = z_1^u(0) = \triangleright \quad \text{and} \quad z_2^u(0) = 0. \quad (\text{Z0})$$

The crucial idea is that the snapshots for  $t > 0$  can be characterized recursively using two auxiliary functions *inputpos* and *prev*.

Since  $M$  is oblivious, the positions of the tape heads on input  $\langle x, u \rangle$  after  $t$  steps are the same for all  $u$  of length  $p(n)$ . We denote the input head positions by *inputpos*( $t$ ).

For every  $t$  we denote by *prev*( $t$ ) the last step before  $t$  in which the output tape head of  $M$  was in the same cell as in step  $t$ . Due to  $M$ 's obliviousness this is again the same for all  $u$  of length  $p(n)$ . If there is no such previous step, because  $t$  is the first time the cell is reached, we set *prev*( $t$ ) =  $t$ . (This deviates from the textbook, which sets *prev*( $t$ ) = 1.) In the other case we have *prev*( $t$ ) <  $t$ .

Also due to  $M$ 's obliviousness, the halting time on input  $\langle x, u \rangle$  is the same for all  $u$  of length  $p(n)$ , and we denote it by  $T' \leq T(|\langle x, u \rangle|)$ . Thus we have *inputpos*( $t$ )  $\leq T'$  for all  $t$ . If we define the symbol sequence  $y(u) = \triangleright \langle x, u \rangle \square^{T'}$ , the first component of the snapshots is, for arbitrary  $t$ :

$$z_0^u(t) = y(u)_{\text{inputpos}(t)}. \quad (\text{Z1})$$

Next we consider the snapshot components  $z_1^u(t)$  for  $t > 0$ . First consider the case *prev*( $t$ ) <  $t$ ; that is, the last time before  $t$  when  $M$ 's output tape head was in the same cell as in step  $t$  was in step *prev*( $t$ ). The snapshot for step *prev*( $t$ ) has exactly the information needed to calculate the actions of  $M$  at step  $t$ : the symbols read from both tapes and the state which  $M$  is in. In some sort of hybrid notation:

$$z_1^u(t) = (M ! z_2^u(\text{prev}(t))) [z_0^u(\text{prev}(t)), z_1^u(\text{prev}(t))] [. ] 1. \quad (\text{Z2})$$

In the other case, *prev*( $t$ ) =  $t$ , the output tape head has not been in this cell before and is thus reading a blank. It cannot be reading the start symbol because the output tape head was in cell zero at step  $t = 0$  already. Formally:

$$z_1^u(t) = \square. \quad (\text{Z3})$$

The state  $z_2^u(t)$  for  $t > 0$  can be computed from the state  $z_2^u(t-1)$  in the previous step and the symbols  $z_0^u(t-1)$  and  $z_1^u(t-1)$  read in the previous step:

$$z_2^u(t) = fst((M! z_2^u(t-1)) [z_0^u(t-1), z_1^u(t-1)]). \quad (\text{Z4})$$

For a string  $u \in \{\mathbf{0}, \mathbf{1}\}^{p(n)}$  the equations (Z0) – (Z4) uniquely determine all the  $z^u(0), \dots, z^u(T')$ . Conversely, the snapshots for  $u$  satisfy all the equations. Therefore the equations characterize the sequence of snapshots.

The condition that  $M$  halts with the output tape head on  $\mathbf{1}$  can be expressed with snapshots:

$$z_1^u(T') = \mathbf{1}. \quad (\text{Z5})$$

This yields our second equivalent statement:  $x \in L$  iff. there is a  $u \in \{\mathbf{0}, \mathbf{1}\}^{p(n)}$  and a sequence  $z^u(0), \dots, z^u(T')$  satisfying the equations (Z0) – (Z5).

### 3rd equivalence

The length of  $y(u)$  is  $m' := m + 1 + T' = 3 + 2n + 2p(n) + T'$  because  $|\langle x, u \rangle| = m$  plus the start symbol plus the  $T'$  blanks.

For the next equivalence we observe that the strings  $y(u)$  for  $u \in \{\mathbf{0}, \mathbf{1}\}^{p(n)}$  can be characterized as follows. Consider a predicate on strings  $y$ :

$$(\text{Y0}) \quad |y| = m';$$

$$(\text{Y1}) \quad y_0 = \triangleright \text{ (the start symbol);}$$

$$(\text{Y2}) \quad y_{2n+1} = y_{2n+2} = \mathbf{1} \text{ (the separator in the pair encoding);}$$

$$(\text{Y3}) \quad y_{2i+1} = \mathbf{0} \text{ for } i = 0, \dots, n-1 \text{ (the zeros before } x\text{);}$$

$$(\text{Y4}) \quad y_{2n+2+2i+1} = \mathbf{0} \text{ for } i = 0, \dots, p(n)-1 \text{ (the zeros before } u\text{);}$$

$$(\text{Y5}) \quad y_{2n+2p(n)+3+i} = \square \text{ for } i = 0, \dots, T'-1 \text{ (the blanks after the input proper);}$$

$$(\text{Y6}) \quad y_{2i+2} = \begin{cases} \mathbf{0} & \text{if } x_i = \mathbf{0}, \\ \mathbf{1} & \text{otherwise} \end{cases} \text{ for } i = 0, \dots, n-1 \text{ (the bits of } x\text{);}$$

$$(\text{Y7}) \quad y_{2n+4+2i} \in \{\mathbf{0}, \mathbf{1}\} \text{ for } i = 0, \dots, p(n)-1 \text{ (the bits of } u\text{).}$$

Every  $y(u)$  for some  $u$  of length  $p(n)$  satisfies this predicate. Conversely, from a  $y$  satisfying the predicate, a  $u$  of length  $p(n)$  can be extracted such that  $y = y(u)$ .

From that we get the third equivalent statement:  $x \in L$  iff. there is a  $y \in \{0, \dots, G-1\}^{m'}$  with (Y0) – (Y7) and a sequence  $z^u(0), \dots, z^u(T')$  with (Z0) – (Z5).

### 4th equivalence

Each element of  $y$  is a symbol from  $M$ 's alphabet, that is, a number less than  $G$ . The same goes for the first two elements of each snapshot,  $z_0^u(t)$  and  $z_1^u(t)$ . The third element,  $z_2^u(t)$ , is a number less than or equal to the number of states of  $M$ . Let  $H$  be the maximum of  $G$  and the number of states. Every element of  $y$  and of the snapshots can then be represented by a bit string of length  $H$  using *num* (the textbook uses binary, but unary is simpler for us). So we use  $3H$  bits to represent one snapshot. There are  $T' + 1$  snapshots until  $M$  halts. Thus all elements of all snapshots can be represented by a string of length  $3H \cdot (T' + 1)$ . Together with the string of length  $N := H \cdot m'$  for the input tape contents  $y$ , we have a total length of  $N + 3H \cdot (T' + 1)$ .

The equivalence can thus be stated as  $x \in L$  iff. there is a string  $s \in \{\mathbf{0}, \mathbf{1}\}^{N+3H \cdot (T'+1)}$  with certain properties. To write these properties we introduce some intervals:

- $\gamma_i = [iH : (i+1)H]$  for  $i < m'$ ,
- $\zeta_0(t) = [N + 3Ht : N + 3Ht + H]$  for  $t \leq T'$ ,
- $\zeta_1(t) = [N + 3Ht + H : N + 3Ht + 2H]$  for  $t \leq T'$ ,

- $\zeta_2(t) = [N + 3Ht + 2H : N + 3H(t + 1)]$  for  $t \leq T'$ .

These intervals slice the string  $s$  in intervals of length  $H$ . The string  $s$  must satisfy properties analogous to (Y0) – (Y7), which we express using the intervals  $\gamma_i$ :

$$(Y0) \quad |s| = N + 3H(T' + 1)$$

$$(Y1) \quad s(\gamma_0) = \triangleright \text{ (the start symbol);}$$

$$(Y2) \quad s(\gamma_{2n+1}) = s(\gamma_{2n+2}) = \mathbf{1} \text{ (the separator in the pair encoding);}$$

$$(Y3) \quad s(\gamma_{2i+1}) = \mathbf{0} \text{ for } i = 0, \dots, n - 1 \text{ (the zeros before } x\text{);}$$

$$(Y4) \quad s(\gamma_{2n+2+2i+1}) = \mathbf{0} \text{ for } i = 0, \dots, p(n) - 1 \text{ (the zeros before } u\text{);}$$

$$(Y5) \quad s(\gamma_{2n+2p(n)+3+i}) = \square \text{ for } i = 0, \dots, T' - 1 \text{ (the blanks after the input proper);}$$

$$(Y6) \quad s(\gamma_{2i+2}) = \begin{cases} \mathbf{0} & \text{if } x_i = \mathbf{0}, \\ \mathbf{1} & \text{otherwise} \end{cases} \text{ for } i = 0, \dots, n - 1 \text{ (the bits of } x\text{);}$$

$$(Y7) \quad s(\gamma_{2n+4+2i}) \in \{\mathbf{0}, \mathbf{1}\} \text{ for } i = 0, \dots, p(n) - 1 \text{ (the bits of } u\text{).}$$

Moreover the string  $s$  must satisfy (Z0) – (Z5). For these properties we use the  $\zeta$  intervals.

$$(Z0) \quad s(\zeta_0(0)) = s(\zeta_1(0)) = \triangleright \text{ and } s(\zeta_2(0)) = 0,$$

$$(Z1) \quad s(\zeta_0(t)) = s(\gamma_{inputpos(t)}) \text{ for } t = 1, \dots, T',$$

$$(Z2) \quad s(\zeta_1(t)) = (M ! s(\zeta_2(prev(t))) [s(\zeta_0(prev(t))), s(\zeta_1(prev(t)))] [.] \mathbf{1} \text{ for } t = 1, \dots, T' \text{ with } prev(t) < t,$$

$$(Z3) \quad s(\zeta_1(t)) = \square \text{ for } t = 1, \dots, T' \text{ with } prev(t) = t,$$

$$(Z4) \quad s(\zeta_2(t)) = fst ((M ! s(\zeta_2(t - 1)) [s(\zeta_0(t - 1)), s(\zeta_1(t - 1))]) \text{ for } t = 1, \dots, T',$$

$$(Z5) \quad s(\zeta_1(T')) = \mathbf{1}.$$

### 5th equivalence

An assignment is an infinite bit string. For formulas over variables with indices in the interval  $[0 : N + 3H(T' + 1)]$ , only the initial  $N + 3H(T' + 1)$  bits of the assignment are relevant. If we had a CNF formula  $\Phi$  over these variables that is satisfied exactly by the assignments  $\alpha$  for which there is an  $s$  with the above properties and  $\alpha(i) = s_i$  for all  $i < |s|$ , then the existence of such an  $s$  would be equivalent to  $\Phi$  being satisfiable.

Next we construct such a CNF formula.

Most properties are easy to translate using the formulas  $\Psi$  and  $\Upsilon$ . For example,  $s(\gamma_0) = \triangleright$  corresponds to  $\alpha(\gamma_0) = \triangleright$ . A formula that is satisfied exactly by assignments with this property is  $\Psi(\gamma_0, 1)$ . Likewise the property  $s(\gamma_{2n+4+2i}) \in \{\mathbf{0}, \mathbf{1}\}$  corresponds to the CNF formula  $\Upsilon(\gamma_{2n+4+2i})$ .

Property (Y0) corresponds to  $\Phi$  having only variables  $0, \dots, N + 3H(T' + 1) - 1$ . The other (Y $\cdot$ ) properties become:

$$(Y1) \quad \Phi_1 := \Psi(\gamma_0, 1),$$

$$(Y2) \quad \Phi_2 := \Psi(\gamma_{2n+1}, 3) \wedge \Psi(\gamma_{2n+2}, 3),$$

$$(Y3) \quad \Phi_3 := \bigwedge_{i=0}^{n-1} \Psi(\gamma_{2i+1}, 2),$$

$$(Y4) \quad \Phi_4 := \bigwedge_{i=0}^{p(n)-1} \Psi(\gamma_{2n+2+2i+1}, 2),$$

$$(Y5) \quad \Phi_5 := \bigwedge_{i=0}^{T'-1} \Psi(\gamma_{2n+2p(n)+3+i}, 0),$$

$$(Y6) \quad \Phi_6 := \bigwedge_{i=0}^{n-1} \Psi(\gamma_{2i+2}, x_i + 2),$$

$$(Y7) \quad \Phi_7 := \bigwedge_{i=0}^{p(n)-1} \Upsilon(\gamma_{2n+4+2i}).$$

Property (Z0) and property (Z5) become these formulas:

$$(Z0) \quad \Phi_0 := \Psi(\zeta_0(0), 1) \wedge \Psi(\zeta_1(0), 1) \wedge \Psi(\zeta_2(0), 0),$$

$$(Z5) \quad \Phi_8 := \Psi(\zeta_1(T'), 3).$$

The remaining properties (Z1) – (Z4) are more complex. They apply to  $t = 1, \dots, T'$ . Let us first consider the case  $prev(t) < t$ . With  $\alpha$  for  $s$  the properties become:

$$(Z1) \quad \alpha(\zeta_0(t)) = \alpha(\gamma_{inputpos}(t)),$$

$$(Z2) \quad \alpha(\zeta_1(t)) = ((M ! \alpha(\zeta_2(prev(t)))) [\alpha(\zeta_0(prev(t))), \alpha(\zeta_1(prev(t)))] [.] 1,$$

$$(Z4) \quad \alpha(\zeta_2(t)) = fst ((M ! \alpha(\zeta_2(t-1))) [\alpha(\zeta_0(t-1)), \alpha(\zeta_1(t-1))]).$$

For any  $t$  the properties (Z1), (Z2), (Z4) use at most  $10H$  variable indices, namely all the variable indices in the nine  $\zeta$ 's and in  $\gamma_{inputpos}(t)$ , each of which have  $H$  indices.

Now if the set of all these variable indices was  $\{0, \dots, 10H - 1\}$  we could apply lemma *depon-ex-formula* to get a CNF formula  $\psi$  over these variables that “captures the spirit” of the properties. We would then merely have to relabel the formula with the actual variable indices we need for each  $t$ . More precisely, let  $w_i = [iH : (i+1)H]$  for  $i = 0, \dots, 9$  and consider the following criterion for  $\alpha$  on the variable indices  $\{0, \dots, 10H - 1\}$ :

$$(F_1) \quad \alpha(w_6) = \alpha(w_9),$$

$$(F_2) \quad \alpha(w_7) = ((M ! \alpha(w_5)) [\alpha(w_3), \alpha(w_4)] [.] 1,$$

$$(F_3) \quad \alpha(w_8) = fst ((M ! \alpha(w_2)) [\alpha(w_0), \alpha(w_1)]).$$

Lemma *depon-ex-formula* gives us a formula  $\psi$  satisfied exactly by those assignments  $\alpha$  that meet the conditions (F<sub>1</sub>), (F<sub>2</sub>), (F<sub>3</sub>). From this  $\psi$  we can create all the formulas we need for representing the properties (Z1), (Z2), (Z4) by substituting (“relabeling” in our terminology) the variables  $[0, 10H)$  appropriately. The substitution for step  $t$  is

$$\varrho_t = \zeta_0(t-1) \circ \zeta_1(t-1) \circ \zeta_2(t-1) \circ \zeta_0(prev(t)) \circ \zeta_1(prev(t)) \circ \zeta_2(prev(t)) \circ \zeta_0(t) \circ \zeta_1(t) \circ \zeta_2(t) \circ \gamma_{inputpos}(t),$$

where  $\circ$  denotes the concatenation of lists. Then  $\varrho_t(\psi)$  is CNF formula satisfied exactly by the assignments  $\alpha$  satisfying (Z1), (Z2), (Z4).

For the case  $prev(t) = t$  we have a criterion on the variable indices  $\{0, \dots, 7H - 1\}$ :

$$(F'_1) \quad \alpha(w_3) = \alpha(w_6),$$

$$(F'_2) \quad \alpha(w_4) = \square,$$

$$(F'_3) \quad \alpha(w_5) = fst ((M ! \alpha(w_2)) [\alpha(w_0), \alpha(w_1)]),$$

whence lemma *depon-ex-formula* supplies us with a formula  $\psi'$ . With appropriate substitutions

$$\varrho'_t = \zeta_0(t-1) \circ \zeta_1(t-1) \circ \zeta_2(t-1) \circ \zeta_0(t) \circ \zeta_1(t) \circ \zeta_2(t) \circ \gamma_{inputpos}(t),$$

we then define CNF formulas  $\chi_t$  for all  $t = 1, \dots, T'$ :

$$\chi_t = \begin{cases} \varrho_t(\psi) & \text{if } prev(t) < t, \\ \varrho'_t(\psi') & \text{if } prev(t) = t. \end{cases}$$

The point of all that is that we can hard-code  $\psi$  and  $\psi'$  in the TM performing the reduction (to be built in the final chapter) and for each  $t$  the TM only needs to construct the substitution  $\varrho_t$  or  $\varrho'_t$  and perform the relabeling. Turing machines that perform these operations will be constructed in the next chapter.

Since all  $\chi_t$  are in CNF, so is the conjunction

$$\Phi_9 := \bigwedge_{t=1}^{T'} \chi_t.$$

Finally the complete CNF formula is:

$$\Phi := \Phi_0 \wedge \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Phi_4 \wedge \Phi_5 \wedge \Phi_6 \wedge \Phi_7 \wedge \Phi_8 \wedge \Phi_9.$$

## 6.2 Auxiliary CNF formulas

In this section we define the CNF formula families  $\Psi$  and  $\Upsilon$ . In the introduction both families were parameterized by intervals of natural numbers. Here we generalize the definition to allow arbitrary sequences of numbers although we will not need this generalization.

The number of variables set to true in a list of variables:

**definition** *numtrue* :: *assignment*  $\Rightarrow$  *nat list*  $\Rightarrow$  *nat* **where**  
*numtrue*  $\alpha$  *vs*  $\equiv$  *length* (*filter*  $\alpha$  *vs*)

Checking whether the list of bits assigned to a list *vs* of variables has the form  $\mathbf{I} \dots \mathbf{I0} \dots \mathbf{0}$ :

**definition** *blocky* :: *assignment*  $\Rightarrow$  *nat list*  $\Rightarrow$  *nat*  $\Rightarrow$  *bool* **where**  
*blocky*  $\alpha$  *vs* *k*  $\equiv$   $\forall i < \text{length } vs. \alpha (vs ! i) \longleftrightarrow i < k$

The next function represents the notation  $\alpha(\gamma)$  from the introduction, albeit generalized to lists that are not intervals  $\gamma$ .

**definition** *unary* :: *assignment*  $\Rightarrow$  *nat list*  $\Rightarrow$  *nat* **where**  
*unary*  $\alpha$  *vs*  $\equiv$  *if* ( $\exists k. \text{blocky } \alpha \text{ vs } k$ ) *then numtrue*  $\alpha$  *vs* *else Suc* (*length vs*)

**lemma** *numtrue-remap*:  
**assumes**  $\forall s \in \text{set } seq. s < \text{length } \sigma$   
**shows** *numtrue* (*remap*  $\sigma$   $\alpha$ ) *seq* = *numtrue*  $\alpha$  (*reseq*  $\sigma$  *seq*)  
*<proof>*

**lemma** *unary-remap*:  
**assumes**  $\forall s \in \text{set } seq. s < \text{length } \sigma$   
**shows** *unary* (*remap*  $\sigma$   $\alpha$ ) *seq* = *unary*  $\alpha$  (*reseq*  $\sigma$  *seq*)  
*<proof>*

Now we define the family  $\Psi$  of CNF formulas. It is parameterized by a list *vs* of variable indices and a number  $k \leq |vs|$ . The formula is satisfied exactly by those assignments that set to true the first *k* variables in *vs* and to false the other variables. This is more general than we need, because for us *vs* will always be an interval.

**definition** *Psi* :: *nat list*  $\Rightarrow$  *nat*  $\Rightarrow$  *formula* ( $\langle \Psi \rangle$ ) **where**  
 $\Psi$  *vs* *k*  $\equiv$  *map* ( $\lambda s. [\text{Pos } s]$ ) (*take* *k vs*) @ *map* ( $\lambda s. [\text{Neg } s]$ ) (*drop* *k vs*)

**lemma** *Psi-unary*:  
**assumes**  $k \leq \text{length } vs$  **and**  $\alpha \models \Psi$  *vs* *k*  
**shows** *unary*  $\alpha$  *vs* = *k*  
*<proof>*

We will only ever consider cases where  $k \leq |vs|$ . So we can use *blocky* to show that an assignment satisfies a  $\Psi$  formula.

**lemma** *satisfies-Psi*:  
**assumes**  $k \leq \text{length } vs$  **and** *blocky*  $\alpha$  *vs* *k*  
**shows**  $\alpha \models \Psi$  *vs* *k*  
*<proof>*

**lemma** *blocky-imp-unary*:  
**assumes**  $k \leq \text{length } vs$  **and** *blocky*  $\alpha$  *vs* *k*  
**shows** *unary*  $\alpha$  *vs* = *k*  
*<proof>*

The family  $\Upsilon$  of CNF formulas also takes as parameter a list of variable indices.

**definition** *Upsilon* :: *nat list*  $\Rightarrow$  *formula* ( $\langle \Upsilon \rangle$ ) **where**  
 $\Upsilon$  *vs*  $\equiv$  *map* ( $\lambda s. [\text{Pos } s]$ ) (*take* 2 *vs*) @ *map* ( $\lambda s. [\text{Neg } s]$ ) (*drop* 3 *vs*)

For  $|vs| > 2$ , an assignment satisfies  $\Upsilon(vs)$  iff. it satisfies  $\Psi(vs, 2)$  or  $\Psi(vs, 3)$ .

**lemma** *Psi-2-imp-Upsilon*:  
**fixes**  $\alpha$  :: *assignment*  
**assumes**  $\alpha \models \Psi$  *vs* 2 **and** *length vs* > 2

**shows**  $\alpha \models \Upsilon$  vs  
 ⟨proof⟩

**lemma** *Psi-3-imp-Upsilon*:  
**assumes**  $\alpha \models \Psi$  vs  $\beta$  **and**  $\text{length } vs > 2$   
**shows**  $\alpha \models \Upsilon$  vs  
 ⟨proof⟩

**lemma** *Upsilon-imp-Psi-2-or-3*:  
**assumes**  $\alpha \models \Upsilon$  vs **and**  $\text{length } vs > 2$   
**shows**  $\alpha \models \Psi$  vs  $2 \vee \alpha \models \Psi$  vs  $3$   
 ⟨proof⟩

**lemma** *Upsilon-unary*:  
**assumes**  $\alpha \models \Upsilon$  vs **and**  $\text{length } vs > 2$   
**shows**  $\text{unary } \alpha$  vs  $= 2 \vee \text{unary } \alpha$  vs  $= 3$   
 ⟨proof⟩

### 6.3 The functions *inputpos* and *prev*

Sequences of the symbol  $\mathbf{0}$ :

**definition** *zeros* ::  $\text{nat} \Rightarrow \text{symbol list}$  **where**  
 $\text{zeros } n \equiv \text{string-to-symbols } (\text{replicate } n \ \mathbf{0})$

**lemma** *length-zeros* [simp]:  $\text{length } (\text{zeros } n) = n$   
 ⟨proof⟩

**lemma** *bit-symbols-zeros*:  $\text{bit-symbols } (\text{zeros } n)$   
 ⟨proof⟩

**lemma** *zeros*:  $\text{zeros } n = \text{replicate } n \ \mathbf{0}$   
 ⟨proof⟩

The assumptions in the following locale are the conditions that according to lemma *NP-imp-oblivious-2tape* hold for all  $\mathcal{NP}$  languages. The construction of  $\Phi$  will take place inside this locale, which in later chapters will be extended to contain the Turing machine outputting  $\Phi$  and the correctness proof for this Turing machine.

**locale** *reduction-sat* =  
**fixes**  $L$  :: *language*  
**fixes**  $M$  :: *machine*  
**and**  $G$  :: *nat*  
**and**  $T$   $p$  ::  $\text{nat} \Rightarrow \text{nat}$   
**assumes**  $T$ : *big-oh-poly*  $T$   
**assumes**  $p$ : *polynomial*  $p$   
**assumes**  $\text{tm-}M$ : *turing-machine*  $2$   $G$   $M$   
**and**  $\text{oblivious-}M$ : *oblivious*  $M$   
**and**  $T\text{-halt}$ :  $\bigwedge y. \text{bit-symbols } y \implies \text{fst } (\text{execute } M \ (\text{start-config } 2 \ y) \ (T \ (\text{length } y))) = \text{length } M$   
**and**  $\text{cert}$ :  $\bigwedge x.$   
 $x \in L \iff (\exists u. \text{length } u = p \ (\text{length } x) \wedge \text{execute } M \ (\text{start-config } 2 \ \langle x; u \rangle) \ (T \ (\text{length } \langle x; u \rangle)) <.> 1 = \mathbf{1})$   
**begin**

The value  $H$  is an upper bound for the number of states of  $M$  and the alphabet size of  $M$ .

**definition**  $H$  :: *nat* **where**  
 $H \equiv \max G \ (\text{length } M)$

**lemma** *H-ge-G*:  $H \geq G$   
 ⟨proof⟩

**lemma** *H-gr-2*:  $H > 2$   
 ⟨proof⟩

**lemma** *H-ge-3*:  $H \geq 3$   
 ⟨*proof*⟩

**lemma** *H-ge-length-M*:  $H \geq \text{length } M$   
 ⟨*proof*⟩

The number of symbols used for encoding one snapshot is  $Z = 3H$ :

**definition** *Z* :: *nat* **where**  
 $Z \equiv 3 * H$

The configuration after running  $M$  on input  $y$  for  $t$  steps:

**abbreviation** *exc* :: *symbol list*  $\Rightarrow$  *nat*  $\Rightarrow$  *config* **where**  
 $\text{exc } y \ t \equiv \text{execute } M \ (\text{start-config } 2 \ y) \ t$

The function  $T$  is just some polynomial upper bound for the running time. The next function,  $TT$ , is the actual running time. Since  $M$  is oblivious, its running time depends only on the length of the input. The argument  $\text{zeros } n$  is thus merely a placeholder for an arbitrary symbol sequence of length  $n$ .

**definition** *TT* :: *nat*  $\Rightarrow$  *nat* **where**  
 $TT \ n \equiv \text{LEAST } t. \text{fst } (\text{exc } (\text{zeros } n) \ t) = \text{length } M$

**lemma** *TT*:  $\text{fst } (\text{exc } (\text{zeros } n) \ (TT \ n)) = \text{length } M$   
 ⟨*proof*⟩

**lemma** *TT-le*:  $TT \ n \leq T \ n$   
 ⟨*proof*⟩

**lemma** *less-TT*:  $t < TT \ n \implies \text{fst } (\text{exc } (\text{zeros } n) \ t) < \text{length } M$   
 ⟨*proof*⟩

**lemma** *oblivious-halt-state*:  
**assumes** *bit-symbols*  $zs$   
**shows**  $\text{fst } (\text{exc } \text{zeros } t) < \text{length } M \longleftrightarrow \text{fst } (\text{exc } (\text{zeros } (\text{length } zs)) \ t) < \text{length } M$   
 ⟨*proof*⟩

**corollary** *less-TT'*:  
**assumes** *bit-symbols*  $zs$  **and**  $t < TT \ (\text{length } zs)$   
**shows**  $\text{fst } (\text{exc } \text{zeros } t) < \text{length } M$   
 ⟨*proof*⟩

**corollary** *TT'*:  
**assumes** *bit-symbols*  $zs$   
**shows**  $\text{fst } (\text{exc } \text{zeros } (TT \ (\text{length } zs))) = \text{length } M$   
 ⟨*proof*⟩

**lemma** *exc-TT-eq-exc-T*:  
**assumes** *bit-symbols*  $zs$   
**shows**  $\text{exc } \text{zeros } (TT \ (\text{length } zs)) = \text{exc } \text{zeros } (T \ (\text{length } zs))$   
 ⟨*proof*⟩

The position of the input tape head of  $M$  depends only on the length  $n$  of the input and the step  $t$ , at least as long as the input is over the alphabet  $\{0, 1\}$ .

**definition** *inputpos* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**  
 $\text{inputpos } n \ t \equiv \text{exc } (\text{zeros } n) \ t <\#\#> 0$

**lemma** *inputpos-oblivious*:  
**assumes** *bit-symbols*  $zs$   
**shows**  $\text{exc } \text{zeros } t <\#\#> 0 = \text{inputpos } (\text{length } zs) \ t$   
 ⟨*proof*⟩

The position of the tape head on the output tape of  $M$  also depends only on the length  $n$  of the input and the step  $t$ .

**lemma** *oblivious-headpos-1*:

**assumes** *bit-symbols zs*  
**shows**  $\text{exc } zs \ t \ \langle \# \rangle \ 1 = \text{exc } (\text{zeros } (\text{length } zs)) \ t \ \langle \# \rangle \ 1$   
*<proof>*

The value  $\text{prev}(t)$  is the most recent step in which  $M$ 's output tape head was in the same position as in step  $t$ . If no such step exists,  $\text{prev}(t)$  is set to  $t$ . Again due to  $M$  being oblivious,  $\text{prev}$  depends only on the length  $n$  of the input (and on  $t$ , of course).

**definition**  $\text{prev} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **where**

$\text{prev } n \ t \equiv$   
*if*  $\exists t' < t. \text{exc } (\text{zeros } n) \ t' \ \langle \# \rangle \ 1 = \text{exc } (\text{zeros } n) \ t \ \langle \# \rangle \ 1$   
*then*  $\text{GREATEST } t'. t' < t \wedge \text{exc } (\text{zeros } n) \ t' \ \langle \# \rangle \ 1 = \text{exc } (\text{zeros } n) \ t \ \langle \# \rangle \ 1$   
*else*  $t$

**lemma** *oblivious-prev*:

**assumes** *bit-symbols zs*  
**shows**  $\text{prev } (\text{length } zs) \ t =$   
*(if*  $\exists t' < t. \text{exc } zs \ t' \ \langle \# \rangle \ 1 = \text{exc } zs \ t \ \langle \# \rangle \ 1$   
*then*  $\text{GREATEST } t'. t' < t \wedge \text{exc } zs \ t' \ \langle \# \rangle \ 1 = \text{exc } zs \ t \ \langle \# \rangle \ 1$   
*else*  $t)$   
*<proof>*

**lemma** *prev-less*:

**assumes**  $\exists t' < t. \text{exc } (\text{zeros } n) \ t' \ \langle \# \rangle \ 1 = \text{exc } (\text{zeros } n) \ t \ \langle \# \rangle \ 1$   
**shows**  $\text{prev } n \ t < t \wedge \text{exc } (\text{zeros } n) \ (\text{prev } n \ t) \ \langle \# \rangle \ 1 = \text{exc } (\text{zeros } n) \ t \ \langle \# \rangle \ 1$   
*<proof>*

**corollary** *prev-less'*:

**assumes** *bit-symbols zs*  
**assumes**  $\exists t' < t. \text{exc } zs \ t' \ \langle \# \rangle \ 1 = \text{exc } zs \ t \ \langle \# \rangle \ 1$   
**shows**  $\text{prev } (\text{length } zs) \ t < t \wedge \text{exc } zs \ (\text{prev } (\text{length } zs) \ t) \ \langle \# \rangle \ 1 = \text{exc } zs \ t \ \langle \# \rangle \ 1$   
*<proof>*

**lemma** *prev-greatest*:

**assumes**  $t' < t$  **and**  $\text{exc } (\text{zeros } n) \ t' \ \langle \# \rangle \ 1 = \text{exc } (\text{zeros } n) \ t \ \langle \# \rangle \ 1$   
**shows**  $t' \leq \text{prev } n \ t$   
*<proof>*

**corollary** *prev-greatest'*:

**assumes** *bit-symbols zs*  
**assumes**  $t' < t$  **and**  $\text{exc } zs \ t' \ \langle \# \rangle \ 1 = \text{exc } zs \ t \ \langle \# \rangle \ 1$   
**shows**  $t' \leq \text{prev } (\text{length } zs) \ t$   
*<proof>*

**lemma** *prev-eq*:  $\text{prev } n \ t = t \iff \neg (\exists t' < t. \text{exc } (\text{zeros } n) \ t' \ \langle \# \rangle \ 1 = \text{exc } (\text{zeros } n) \ t \ \langle \# \rangle \ 1)$   
*<proof>*

**lemma** *prev-le*:  $\text{prev } n \ t \leq t$   
*<proof>*

**corollary** *prev-eq'*:

**assumes** *bit-symbols zs*  
**shows**  $\text{prev } (\text{length } zs) \ t = t \iff \neg (\exists t' < t. \text{exc } zs \ t' \ \langle \# \rangle \ 1 = \text{exc } zs \ t \ \langle \# \rangle \ 1)$   
*<proof>*

**lemma** *prev-between*:

**assumes**  $\text{prev } n \ t < t'$  **and**  $t' < t$   
**shows**  $\text{exc } (\text{zeros } n) \ t' \ \langle \# \rangle \ 1 \neq \text{exc } (\text{zeros } n) \ (\text{prev } n \ t) \ \langle \# \rangle \ 1$   
*<proof>*

**lemma** *prev-write-read*:

**assumes** *bit-symbols zs* **and**  $n = \text{length } zs$   
**and**  $\text{prev } n \ t < t$  **and**  $\text{cfg} = \text{exc } zs \ (\text{prev } n \ t)$  **and**  $t \leq TT \ n$   
**shows**  $\text{exc } zs \ t \ \langle \cdot \rangle \ 1 = (M ! (\text{fst } \text{cfg})) [\text{cfg} \ \langle \cdot \rangle \ 0, \text{cfg} \ \langle \cdot \rangle \ 1] [] \ 1$

*<proof>*

**lemma** *prev-no-write*:

**assumes** *bit-symbols zs* **and**  $n = \text{length } zs$   
**and**  $\text{prev } n \ t = t$  **and**  $t \leq TT \ n$  **and**  $t > 0$   
**shows**  $\text{exc } zs \ t <.> 1 = \square$

*<proof>*

The intervals  $\gamma_i$  and  $w_0, \dots, w_9$  do not depend on  $x$ , and so can be defined here already.

**definition** *gamma* ::  $\text{nat} \Rightarrow \text{nat list}$  ( $\langle \gamma \rangle$ ) **where**

$\gamma \ i \equiv [i * H .. < \text{Suc } i * H]$

**lemma** *length-gamma* [*simp*]:  $\text{length } (\gamma \ i) = H$

*<proof>*

**abbreviation**  $w_0 \equiv [0 .. < H]$

**abbreviation**  $w_1 \equiv [H .. < 2 * H]$

**abbreviation**  $w_2 \equiv [2 * H .. < Z]$

**abbreviation**  $w_3 \equiv [Z .. < Z + H]$

**abbreviation**  $w_4 \equiv [Z + H .. < Z + 2 * H]$

**abbreviation**  $w_5 \equiv [Z + 2 * H .. < 2 * Z]$

**abbreviation**  $w_6 \equiv [2 * Z .. < 2 * Z + H]$

**abbreviation**  $w_7 \equiv [2 * Z + H .. < 2 * Z + 2 * H]$

**abbreviation**  $w_8 \equiv [2 * Z + 2 * H .. < 3 * Z]$

**abbreviation**  $w_9 \equiv [3 * Z .. < 3 * Z + H]$

**lemma** *unary-upt-eq*:

**fixes**  $\alpha_1 \ \alpha_2$  :: *assignment*

**and** *lower upper k* :: *nat*

**assumes**  $\forall i < k. \alpha_1 \ i = \alpha_2 \ i$  **and**  $\text{upper} \leq k$

**shows**  $\text{unary } \alpha_1 \ [\text{lower} .. < \text{upper}] = \text{unary } \alpha_2 \ [\text{lower} .. < \text{upper}]$

*<proof>*

For the case  $\text{prev } m \ t < t$ , we have the following predicate on assignments, which corresponds to  $(F_1)$ ,  $(F_2)$ ,  $(F_3)$  from the introduction:

**definition** *F* :: *assignment*  $\Rightarrow$  *bool* **where**

$F \ \alpha \equiv$

$\text{unary } \alpha \ w_6 = \text{unary } \alpha \ w_9 \ \wedge$

$\text{unary } \alpha \ w_7 = (M \ ! \ (\text{unary } \alpha \ w_5)) \ [\text{unary } \alpha \ w_3, \ \text{unary } \alpha \ w_4] \ [.] \ 1 \ \wedge$

$\text{unary } \alpha \ w_8 = \text{fst} \ ((M \ ! \ (\text{unary } \alpha \ w_2)) \ [\text{unary } \alpha \ w_0, \ \text{unary } \alpha \ w_1])$

**lemma** *depon-F*:  $\text{depon } (3 * Z + H) \ F$

*<proof>*

There is a CNF formula  $\psi$  that contains the first  $3Z + H$  variables and is satisfied by exactly the assignments specified by  $F$ .

**definition** *psi* :: *formula* ( $\langle \psi \rangle$ ) **where**

$\psi \equiv \text{SOME } \varphi.$

$\text{fsize } \varphi \leq (3 * Z + H) * 2 \wedge (3 * Z + H) \wedge$

$\text{length } \varphi \leq 2 \wedge (3 * Z + H) \wedge$

$\text{variables } \varphi \subseteq \{.. < 3 * Z + H\} \wedge$

$(\forall \alpha. F \ \alpha = \alpha \models \varphi)$

**lemma** *psi*:

$\text{fsize } \psi \leq (3 * Z + H) * 2 \wedge (3 * Z + H) \wedge$

$\text{length } \psi \leq 2 \wedge (3 * Z + H) \wedge$

$\text{variables } \psi \subseteq \{.. < 3 * Z + H\} \wedge$

$(\forall \alpha. F \ \alpha = \alpha \models \psi)$

*<proof>*

**lemma** *satisfies-psi*:

**assumes**  $\text{length } \sigma = 3 * Z + H$

**shows**  $\alpha \models \text{relabel } \sigma \ \psi = \text{remap } \sigma \ \alpha \models \psi$

*<proof>*

**lemma** *psi-F*:  $\text{remap } \sigma \ \alpha \models \psi = F (\text{remap } \sigma \ \alpha)$   
*<proof>*

**corollary** *satisfies-F*:

**assumes**  $\text{length } \sigma = 3 * Z + H$

**shows**  $\alpha \models \text{relabel } \sigma \ \psi = F (\text{remap } \sigma \ \alpha)$

*<proof>*

For the case  $\text{prev } m \ t = t$ , the following predicate corresponds to  $(F'_1)$ ,  $(F'_2)$ ,  $(F'_3)$  from the introduction:

**definition**  $F' :: \text{assignment} \Rightarrow \text{bool}$  **where**

$F' \ \alpha \equiv$

$\text{unary } \alpha \ w_3 = \text{unary } \alpha \ w_6 \ \wedge$

$\text{unary } \alpha \ w_4 = 0 \ \wedge$

$\text{unary } \alpha \ w_5 = \text{fst } ((M ! (\text{unary } \alpha \ w_2)) [\text{unary } \alpha \ w_0, \text{unary } \alpha \ w_1])$

**lemma** *depon-F'*:  $\text{depon } (2 * Z + H) \ F'$

*<proof>*

The CNF formula  $\psi'$  is analogous to  $\psi$  from the previous case.

**definition**  $\text{psi}' :: \text{formula } (\langle \psi' \rangle)$  **where**

$\psi' \equiv \text{SOME } \varphi.$

$\text{fsize } \varphi \leq (2 * Z + H) * 2 \wedge (2 * Z + H) \ \wedge$

$\text{length } \varphi \leq 2 \wedge (2 * Z + H) \ \wedge$

$\text{variables } \varphi \subseteq \{.. < 2 * Z + H\} \ \wedge$

$(\forall \alpha. F' \ \alpha = \alpha \models \varphi)$

**lemma** *psi'*:

$\text{fsize } \psi' \leq (2 * Z + H) * 2 \wedge (2 * Z + H) \ \wedge$

$\text{length } \psi' \leq 2 \wedge (2 * Z + H) \ \wedge$

$\text{variables } \psi' \subseteq \{.. < 2 * Z + H\} \ \wedge$

$(\forall \alpha. F' \ \alpha = \alpha \models \psi')$

*<proof>*

**lemma** *satisfies-psi'*:

**assumes**  $\text{length } \sigma = 2 * Z + H$

**shows**  $\alpha \models \text{relabel } \sigma \ \psi' = \text{remap } \sigma \ \alpha \models \psi'$

*<proof>*

**lemma** *psi'-F'*:  $\text{remap } \sigma \ \alpha \models \psi' = F' (\text{remap } \sigma \ \alpha)$

*<proof>*

**corollary** *satisfies-F'*:

**assumes**  $\text{length } \sigma = 2 * Z + H$

**shows**  $\alpha \models \text{relabel } \sigma \ \psi' = F' (\text{remap } \sigma \ \alpha)$

*<proof>*

**end**

## 6.4 Snapshots

The snapshots and much of the rest of the construction of  $\Phi$  depend on the string  $x$ . We encapsulate this in a sublocale of *reduction-sat*.

**locale** *reduction-sat-x* = *reduction-sat* +

**fixes**  $x :: \text{string}$

**begin**

**abbreviation**  $n :: \text{nat}$  **where**

$n \equiv \text{length } x$

Turing machines consume the string  $x$  as a sequence  $xs$  of symbols:

**abbreviation**  $xs :: \text{symbol list}$  **where**

$xs \equiv \text{string-to-symbols } x$

**lemma**  $bs-xs$ : *bit-symbols*  $xs$

*<proof>*

For the verifier Turing machine  $M$  we are only concerned with inputs of the form  $\langle x, u \rangle$  for a string  $u$  of length  $p(n)$ . The pair  $\langle x, u \rangle$  has the length  $m = 2n + 2p(n) + 2$ .

**definition**  $m :: \text{nat}$  **where**

$m \equiv 2 * n + 2 * p \ n + 2$

On input  $\langle x, u \rangle$  the Turing machine  $M$  halts after  $T' = TT(m)$  steps.

**definition**  $T' :: \text{nat}$  **where**

$T' \equiv TT \ m$

The positions of both of  $M$ 's tape heads are bounded by  $T'$ .

**lemma** *inputpos-less*: *inputpos*  $m \ t \leq T'$

*<proof>*

**lemma** *headpos-1-less*: *exc* (*zeros*  $m$ )  $t <\#\> 1 \leq T'$

*<proof>*

The formula  $\Phi$  must contain a condition for every symbol that  $M$  is reading from the input tape. While  $T'$  is an upper bound for the input tape head position of  $M$ , it might be that  $T'$  is less than the length of the input  $\langle x, u \rangle$ . So the portion of the input read by  $M$  might be a prefix of the input or it might be the input followed by some blanks afterwards. This would make for an awkward case distinction. We do not have to be very precise here and can afford to bound the portion of the input tape read by  $M$  by the number  $m' = 2n + 2p(n) + 3 + T'$ , which is the length of the start symbol followed by the input  $\langle x, u \rangle$  followed by  $T'$  blanks. This symbol sequence was called  $y(u)$  in the introduction. Here we will call it *ysymbols*  $u$ .

**definition**  $m' :: \text{nat}$  **where**

$m' \equiv 2 * n + 2 * p \ n + 3 + T'$

**definition** *ysymbols*  $:: \text{string} \Rightarrow \text{symbol list}$  **where**

*ysymbols*  $u \equiv 1 \ \# \ \langle x; u \rangle \ @ \ \text{replicate } T' \ 0$

**lemma** *length-ysymbols*:  $\text{length } u = p \ n \implies \text{length } (\text{ysymbols } u) = m'$

*<proof>*

**lemma** *ysymbols-init*:

**assumes**  $i < \text{length } (\text{ysymbols } u)$

**shows**  $\text{ysymbols } u \ ! \ i = (\text{start-config } 2 \ \langle x; u \rangle \ <:\> \ 0) \ i$

*<proof>*

**lemma** *ysymbols-at-0*:  $\text{ysymbols } u \ ! \ 0 = 1$

*<proof>*

**lemma** *ysymbols-first-at*:

**assumes**  $j < \text{length } x$

**shows**  $\text{ysymbols } u \ ! \ (2*j+1) = 2$

**and**  $\text{ysymbols } u \ ! \ (2*j+2) = (\text{if } x \ ! \ j \ \text{then } 3 \ \text{else } 2)$

*<proof>*

**lemma** *ysymbols-at-2n1*:  $\text{ysymbols } u \ ! \ (2*n+1) = 3$

*<proof>*

**lemma** *ysymbols-at-2n2*:  $\text{ysymbols } u \ ! \ (2*n+2) = 3$

*<proof>*

**lemma** *ysymbols-second-at*:

**assumes**  $j < \text{length } u$

**shows**  $\text{ysymbols } u \ ! \ (2*n+2+2*j+1) = 2$

**and**  $ysymbols\ u\ !\ (2*n+2+2*j+2) = (if\ u\ !\ j\ then\ 3\ else\ 2)$   
 ⟨proof⟩

**lemma** *ysymbols-last*:

**assumes**  $length\ u = p\ n$  **and**  $i > m$  **and**  $i < m + 1 + T'$

**shows**  $ysymbols\ u\ !\ i = 0$

⟨proof⟩

The number of symbols used for unary encoding  $m'$  symbols will be called  $N$ :

**definition**  $N :: nat$  **where**

$N \equiv H * m'$

**lemma** *N-eq*:  $N = H * (2 * n + 2 * p\ n + 3 + T')$

⟨proof⟩

**lemma** *m'*:  $m' * H = N$

⟨proof⟩

**lemma** *inputpos-less'*:  $inputpos\ m\ t < m'$

⟨proof⟩

**lemma** *T'-less*:  $T' < N$

⟨proof⟩

The three components of a snapshot:

**definition**  $z0 :: string \Rightarrow nat \Rightarrow symbol$  **where**

$z0\ u\ t \equiv exc\ \langle x; u \rangle\ t\ <.>\ 0$

**definition**  $z1 :: string \Rightarrow nat \Rightarrow symbol$  **where**

$z1\ u\ t \equiv exc\ \langle x; u \rangle\ t\ <.>\ 1$

**definition**  $z2 :: string \Rightarrow nat \Rightarrow state$  **where**

$z2\ u\ t \equiv fst\ (exc\ \langle x; u \rangle\ t)$

**lemma** *z0-le*:  $z0\ u\ t \leq H$

⟨proof⟩

**lemma** *z1-le*:  $z1\ u\ t \leq H$

⟨proof⟩

**lemma** *z2-le*:  $z2\ u\ t \leq H$

⟨proof⟩

The next lemma corresponds to (Z1) from the second equivalence mentioned in the introduction. It expresses the first element of a snapshot in terms of  $y(u)$  and  $inputpos$ .

**lemma** *z0*:

**assumes**  $length\ u = p\ n$

**shows**  $z0\ u\ t = ysymbols\ u\ !\ (inputpos\ m\ t)$

⟨proof⟩

The next lemma corresponds to (Z2) from the second equivalence mentioned in the introduction. It shows how, in the case  $prev(t) < t$ , the second component of a snapshot can be expressed recursively using snapshots for earlier steps.

**lemma** *z1*:

**assumes**  $length\ u = p\ n$  **and**  $prev\ m\ t < t$  **and**  $t \leq T'$

**shows**  $z1\ u\ t = (M\ !\ (z2\ u\ (prev\ m\ t)))\ [z0\ u\ (prev\ m\ t),\ z1\ u\ (prev\ m\ t)]\ [.] 1$

⟨proof⟩

The next lemma corresponds to (Z3) from the second equivalence mentioned in the introduction. It shows that in the case  $prev(t) = t$ , the second component of a snapshot equals the blank symbol.

**lemma** *z1'*:

**assumes**  $length\ u = p\ n$  **and**  $prev\ m\ t = t$  **and**  $0 < t$  **and**  $t \leq T'$

**shows**  $z1\ u\ t = \square$   
 ⟨proof⟩

The next lemma corresponds to (Z4) from the second equivalence mentioned in the introduction. It shows how the third component of a snapshot can be expressed recursively using snapshots for earlier steps.

**lemma**  $z2$ :  
**assumes**  $length\ u = p\ n$  **and**  $t < T'$   
**shows**  $z2\ u\ (Suc\ t) = fst\ ((M\ !\ (z2\ u\ t))\ [z0\ u\ t,\ z1\ u\ t])$   
 ⟨proof⟩

**corollary**  $z2'$ :  
**assumes**  $length\ u = p\ n$  **and**  $t > 0$  **and**  $t \leq T'$   
**shows**  $z2\ u\ t = fst\ ((M\ !\ (z2\ u\ (t - 1)))\ [z0\ u\ (t - 1),\ z1\ u\ (t - 1)])$   
 ⟨proof⟩

The intervals  $\zeta_0$ ,  $\zeta_1$ , and  $\zeta_2$  are long enough for a unary encoding of the three components of a snapshot:

**definition**  $zeta0 :: nat \Rightarrow nat\ list\ (\langle \zeta_0 \rangle)$  **where**  
 $\zeta_0\ t \equiv [N + t * Z .. N + t * Z + H]$

**definition**  $zeta1 :: nat \Rightarrow nat\ list\ (\langle \zeta_1 \rangle)$  **where**  
 $\zeta_1\ t \equiv [N + t * Z + H .. N + t * Z + 2 * H]$

**definition**  $zeta2 :: nat \Rightarrow nat\ list\ (\langle \zeta_2 \rangle)$  **where**  
 $\zeta_2\ t \equiv [N + t * Z + 2 * H .. N + (Suc\ t) * Z]$

**lemma**  $length\text{-}zeta0$  [simp]:  $length\ (\zeta_0\ t) = H$   
 ⟨proof⟩

**lemma**  $length\text{-}zeta1$  [simp]:  $length\ (\zeta_1\ t) = H$   
 ⟨proof⟩

**lemma**  $length\text{-}zeta2$  [simp]:  $length\ (\zeta_2\ t) = H$   
 ⟨proof⟩

The substitutions  $\varrho_t$ , which have to be applied to  $\psi$  to get the CNF formulas  $\chi_t$  for the case  $prev(t) < t$ :

**definition**  $\rho :: nat \Rightarrow nat\ list\ (\langle \varrho \rangle)$  **where**  
 $\varrho\ t \equiv$   
 $\zeta_0\ (t - 1)\ @\ \zeta_1\ (t - 1)\ @\ \zeta_2\ (t - 1)\ @$   
 $\zeta_0\ (prev\ m\ t)\ @\ \zeta_1\ (prev\ m\ t)\ @\ \zeta_2\ (prev\ m\ t)\ @$   
 $\zeta_0\ t\ @\ \zeta_1\ t\ @\ \zeta_2\ t\ @$   
 $\gamma\ (inputpos\ m\ t)$

**lemma**  $length\text{-}\rho$ :  $length\ (\varrho\ t) = 3 * Z + H$   
 ⟨proof⟩

The substitutions  $\varrho'_t$ , which have to be applied to  $\psi'$  to get the CNF formulas  $\chi_t$  for the case  $prev(t) = t$ :

**definition**  $\rho' :: nat \Rightarrow nat\ list\ (\langle \varrho' \rangle)$  **where**  
 $\varrho'\ t \equiv$   
 $\zeta_0\ (t - 1)\ @\ \zeta_1\ (t - 1)\ @\ \zeta_2\ (t - 1)\ @$   
 $\zeta_0\ t\ @\ \zeta_1\ t\ @\ \zeta_2\ t\ @$   
 $\gamma\ (inputpos\ m\ t)$

**lemma**  $length\text{-}\rho'$ :  $length\ (\varrho'\ t) = 2 * Z + H$   
 ⟨proof⟩

An auxiliary lemma for accessing the  $n$ -th element of a list sandwiched between two lists. It will be applied to  $xs = \varrho_t$  or  $xs = \varrho'_t$ :

**lemma**  $nth\text{-}append3$ :  
**fixes**  $xs\ ys\ zs\ ws :: 'a\ list$  **and**  $n\ i :: nat$   
**assumes**  $xs = ys\ @\ zs\ @\ ws$  **and**  $i < length\ zs$  **and**  $n = length\ ys$   
**shows**  $xs\ !\ (n + i) = zs\ !\ i$   
 ⟨proof⟩

The formulas  $\chi_t$  representing snapshots for  $0 < t \leq T'$ :

**definition**  $\chi_t :: \text{nat} \Rightarrow \text{formula} \langle \chi_t \rangle$  **where**

$\chi_t \equiv \text{if } \text{prev } m \ t < t \text{ then relabel } (\varrho \ t) \ \psi \text{ else relabel } (\varrho' \ t) \ \psi'$

The crucial feature of the formulas  $\chi_t$  for  $t > 0$  is that they are satisfied by exactly those assignments that represent in their bits  $N$  to  $N + Z \cdot (T' + 1)$  the  $T' + 1$  snapshots of  $M$  on input  $\langle x, u \rangle$  when the relevant portion of the input tape is encoded in the first  $N$  bits of the assignment.

This works because the  $\chi_t$  constrain the assignment to meet the recursive characterizations (Z1) — (Z4) for the snapshots.

The next two lemmas make this more precise. We first consider the case  $\text{prev}(t) < t$ . The following lemma says  $\alpha$  satisfies  $\chi_t$  iff.  $\alpha$  satisfies the properties (Z1), (Z2), and (Z4).

**lemma** *satisfies-chi-less*:

**fixes**  $\alpha :: \text{assignment}$

**assumes**  $\text{prev } m \ t < t$

**shows**  $\alpha \models \chi_t \iff$

$\text{unary } \alpha \ (\zeta_0 \ t) = \text{unary } \alpha \ (\gamma \ (\text{inputpos } m \ t)) \wedge$

$\text{unary } \alpha \ (\zeta_1 \ t) = (M ! (\text{unary } \alpha \ (\zeta_2 \ (\text{prev } m \ t)))) [\text{unary } \alpha \ (\zeta_0 \ (\text{prev } m \ t)), \text{unary } \alpha \ (\zeta_1 \ (\text{prev } m \ t))] [.] \ 1 \wedge$

$\text{unary } \alpha \ (\zeta_2 \ t) = \text{fst} ((M ! (\text{unary } \alpha \ (\zeta_2 \ (t - 1)))) [\text{unary } \alpha \ (\zeta_0 \ (t - 1)), \text{unary } \alpha \ (\zeta_1 \ (t - 1))])$

*<proof>*

Next we consider the case  $\text{prev}(t) = t$ . The following lemma says  $\alpha$  satisfies  $\chi_t$  iff.  $\alpha$  satisfies the properties (Z1), (Z2), and (Z3).

**lemma** *satisfies-chi-eq*:

**assumes**  $\text{prev } m \ t = t$  **and**  $t \leq T'$

**shows**  $\alpha \models \chi_t \iff$

$\text{unary } \alpha \ (\zeta_0 \ t) = \text{unary } \alpha \ (\gamma \ (\text{inputpos } m \ t)) \wedge$

$\text{unary } \alpha \ (\zeta_1 \ t) = 0 \wedge$

$\text{unary } \alpha \ (\zeta_2 \ t) = \text{fst} ((M ! (\text{unary } \alpha \ (\zeta_2 \ (t - 1)))) [\text{unary } \alpha \ (\zeta_0 \ (t - 1)), \text{unary } \alpha \ (\zeta_1 \ (t - 1))])$

*<proof>*

## 6.5 The CNF formula $\Phi$

We can now formulate all the parts  $\Phi_0, \dots, \Phi_9$  of the complete formula  $\Phi$ , and thus  $\Phi$  itself.

Representing the snapshot in step 0:

**definition**  $\Phi_{I0} :: \text{formula} \langle \Phi_{I0} \rangle$  **where**

$\Phi_{I0} \equiv \Psi \ (\zeta_0 \ 0) \ 1 \ @ \ \Psi \ (\zeta_1 \ 0) \ 1 \ @ \ \Psi \ (\zeta_2 \ 0) \ 0$

The start symbol at the beginning of the input tape:

**definition**  $\Phi_{I1} :: \text{formula} \langle \Phi_{I1} \rangle$  **where**

$\Phi_{I1} \equiv \Psi \ (\gamma \ 0) \ 1$

The separator **11** between  $x$  and  $u$ :

**definition**  $\Phi_{I2} :: \text{formula} \langle \Phi_{I2} \rangle$  **where**

$\Phi_{I2} \equiv \Psi \ (\gamma \ (2*n+1)) \ 3 \ @ \ \Psi \ (\gamma \ (2*n+2)) \ 3$

The zeros before the symbols of  $x$ :

**definition**  $\Phi_{I3} :: \text{formula} \langle \Phi_{I3} \rangle$  **where**

$\Phi_{I3} \equiv \text{concat} \ (\text{map} \ (\lambda i. \ \Psi \ (\gamma \ (2*i+1)) \ 2) \ [0..<n])$

The zeros before the symbols of  $u$ :

**definition**  $\Phi_{I4} :: \text{formula} \langle \Phi_{I4} \rangle$  **where**

$\Phi_{I4} \equiv \text{concat} \ (\text{map} \ (\lambda i. \ \Psi \ (\gamma \ (2*n+2+2*i+1)) \ 2) \ [0..<p \ n])$

The blank symbols after the input  $\langle x, u \rangle$ :

**definition**  $\Phi_{I5} :: \text{formula} \langle \Phi_{I5} \rangle$  **where**

$\Phi_{I5} \equiv \text{concat} \ (\text{map} \ (\lambda i. \ \Psi \ (\gamma \ (2*n + 2*p \ n + 3 + i)) \ 0) \ [0..<T'])$

The symbols of  $x$ :

**definition PHI6** :: formula ( $\langle \Phi_6 \rangle$ ) **where**

$$\Phi_6 \equiv \text{concat} (\text{map} (\lambda i. \Psi (\gamma (2*i+2))) (\text{if } x ! i \text{ then } 3 \text{ else } 2)) [0..<n])$$

Constraining the symbols of  $u$  to be from  $\{0, 1\}$ :

**definition PHI7** :: formula ( $\langle \Phi_7 \rangle$ ) **where**

$$\Phi_7 \equiv \text{concat} (\text{map} (\lambda i. \Upsilon (\gamma (2*n+4+2*i))) [0..<p n])$$

Reading a  $1$  in the final step to signal acceptance of  $\langle x, u \rangle$ :

**definition PHI8** :: formula ( $\langle \Phi_8 \rangle$ ) **where**

$$\Phi_8 \equiv \Psi (\zeta_1 T') 3$$

The snapshots after the first and before the last:

**definition PHI9** :: formula ( $\langle \Phi_9 \rangle$ ) **where**

$$\Phi_9 \equiv \text{concat} (\text{map} (\lambda t. \chi (\text{Suc } t)) [0..<T'])$$

The complete formula:

**definition PHI** :: formula ( $\langle \Phi \rangle$ ) **where**

$$\Phi \equiv \Phi_0 @ \Phi_1 @ \Phi_2 @ \Phi_3 @ \Phi_4 @ \Phi_5 @ \Phi_6 @ \Phi_7 @ \Phi_8 @ \Phi_9$$

## 6.6 Correctness of the formula

We have to show that the formula  $\Phi$  is satisfiable if and only if  $x \in L$ . There is a subsection for both of the implications. Instead of  $x \in L$  we will use the right-hand side of the following equivalence.

**lemma L-iff-ex-u:**  $x \in L \longleftrightarrow (\exists u. \text{length } u = p n \wedge \text{exc } \langle x; u \rangle T' <.> 1 = \mathbf{1})$

*<proof>*

### 6.6.1 $\Phi$ satisfiable implies $x \in L$

The proof starts from an assignment  $\alpha$  satisfying  $\Phi$  and shows that there is a string  $u$  of length  $p(n)$  such that  $M$ , on input  $\langle x, u \rangle$ , halts with the output tape head on the symbol  $1$ . The overarching idea is that  $\alpha$ , by satisfying  $\Phi$ , encodes a string  $u$  and a computation of  $M$  on  $u$  that results in  $M$  halting with the output tape head on the symbol  $1$ .

The assignment  $\alpha$  is an infinite bit string, whose first  $N = m' \cdot H$  bits are supposed to encode the first  $m'$  symbols on  $M$ 's input tape, which contains the pair  $\langle x, u \rangle$ . The first step of the proof is thus to extract a  $u$  of length  $p(n)$  from the first  $N$  bits of  $\alpha$ . The Formula  $\Phi_7$  ensures that the symbols representing  $u$  are  $0$  or  $1$  and thus represent a bit string.

Next the proof shows that the first  $N$  bits of  $\alpha$  encode the relevant portion  $y(u)$  of the input tape for the  $u$  just extracted, that is,  $y(u)_i = \alpha(\gamma_i)$  for  $i < m'$ . The proof exploits the constraints set by  $\Phi_1$  to  $\Phi_6$ . In particular this implies that  $y(u)_{\text{inputpos}(t)} = \alpha(\gamma_{\text{inputpos}(t)})$  for all  $t$ .

The next  $Z \cdot (T' + 1)$  bits of  $\alpha$  encode  $T' + 1$  snapshots. More precisely, we prove  $z_0(u, t) = \alpha(\zeta_0^t)$  and  $z_1(u, t) = \alpha(\zeta_1^t)$  and  $z_2(u, t) = \alpha(\zeta_2^t)$  for all  $t \leq T'$ . This works by induction on  $t$ . The case  $t = 0$  is covered by the formula  $\Phi_0$ . For  $0 < t \leq T'$  the formulas  $\chi_t$  are responsible, which make up  $\Phi_9$ . Basically  $\chi_t$  represents the recursive characterization of the snapshot  $z_t$  in terms of earlier snapshots (of  $t - 1$  and possibly  $\text{prev}(t)$ ). This is the trickiest part and we need some preliminary lemmas for that.

Once that is done, we know that some bits of  $\alpha$ , namely  $\alpha(\zeta_1(T'))$ , encode the symbol under the output tape head after  $T'$  steps, that is, when  $M$  has halted. Formula  $\Phi_8$  ensures that this symbol is  $1$ , which concludes the proof.

**lemma sat-PHI-imp-ex-u:**

**assumes** satisfiable  $\Phi$

**shows**  $\exists u. \text{length } u = p n \wedge \text{exc } \langle x; u \rangle T' <.> 1 = \mathbf{1}$

*<proof>*

### 6.6.2 $x \in L$ implies $\Phi$ satisfiable

For the other direction, we assume a string  $x \in L$  and show that the formula  $\Phi$  derived from it is satisfiable. From  $x \in L$  it follows that there is a certificate  $u$  of length  $p(n)$  such that  $M$  on input  $\langle x, u \rangle$  halts after  $T'$  steps with the output tape head on the symbol  $1$ .

An assignment that satisfies  $\Phi$  must have its first  $N = m' \cdot H$  bits set in such a way that they encode the relevant portion  $y(u)$  of the input tape, that is, with  $\langle x, u \rangle$  followed by  $T'$  blanks. This will take care of satisfying  $\Phi_1, \dots, \Phi_7$ . The next  $Z \cdot (T' + 1)$  bits of  $\alpha$  must be set such that they encode the  $T' + 1$  snapshots of  $M$  when started on  $\langle x, u \rangle$ . This way  $\Phi_0$  and  $\Phi_9$  will be satisfied. Finally,  $\Phi_8$  is satisfied because by the choice of  $u$  the last snapshot contains a  $\mathbf{1}$  as the symbol under the output tape head.

The following function maps a string  $u$  to an assignment as just described.

**definition**  $\beta :: \text{string} \Rightarrow \text{assignment} (\langle \beta \rangle)$  **where**

```

 $\beta \ u \ i \equiv$ 
  if  $i < N$  then
    let
       $j = i \text{ div } H;$ 
       $k = i \text{ mod } H$ 
    in
      if  $j = 0$  then  $k < 1$ 
      else if  $j = 2 * n + 1 \vee j = 2 * n + 2$  then  $k < 3$ 
      else if  $j \geq 2 * n + 2 * p \ n + 3$  then  $k < 0$ 
      else if odd  $j$  then  $k < 2$ 
      else if  $j \leq 2 * n$  then  $k < (\text{if } x ! (j \text{ div } 2 - 1) \text{ then } 3 \text{ else } 2)$ 
      else  $k < (\text{if } u ! (j \text{ div } 2 - n - 2) \text{ then } 3 \text{ else } 2)$ 
    else if  $i < N + Z * (\text{Suc } T')$  then
      let  $t = (i - N) \text{ div } Z; k = (i - N) \text{ mod } Z$  in
        if  $k < H$  then  $k < z0 \ u \ t$ 
        else if  $k < 2 * H$  then  $k - H < z1 \ u \ t$ 
        else  $k - 2 * H < z2 \ u \ t$ 
    else False

```

In order to show that  $\beta(u)$  satisfies  $\Phi$ , we show that it satisfies all parts of  $\Phi$ . These parts consist mostly of  $\Psi$  formulas, whose satisfiability can be proved using lemma *satisfies-Psi*. To apply this lemma, the following ones will be helpful.

**lemma** *blocky-gammaI*:

```

assumes  $\bigwedge k. k < H \implies \alpha (j * H + k) = (k < l)$ 
shows blocky  $\alpha (\gamma \ j) \ l$ 
 $\langle \text{proof} \rangle$ 

```

**lemma** *blocky-zeta0I*:

```

assumes  $\bigwedge k. k < H \implies \alpha (N + t * Z + k) = (k < l)$ 
shows blocky  $\alpha (\zeta_0 \ t) \ l$ 
 $\langle \text{proof} \rangle$ 

```

**lemma** *blocky-zeta1I*:

```

assumes  $\bigwedge k. k < H \implies \alpha (N + t * Z + H + k) = (k < l)$ 
shows blocky  $\alpha (\zeta_1 \ t) \ l$ 
 $\langle \text{proof} \rangle$ 

```

**lemma** *blocky-zeta2I*:

```

assumes  $\bigwedge k. k < H \implies \alpha (N + t * Z + 2 * H + k) = (k < l)$ 
shows blocky  $\alpha (\zeta_2 \ t) \ l$ 
 $\langle \text{proof} \rangle$ 

```

**lemma** *beta-1*: *blocky*  $(\beta \ u) (\gamma \ 0) \ 1$

$\langle \text{proof} \rangle$

**lemma** *beta-2a*: *blocky*  $(\beta \ u) (\gamma \ (2 * n + 1)) \ 3$

$\langle \text{proof} \rangle$

**lemma** *beta-2b*: *blocky*  $(\beta \ u) (\gamma \ (2 * n + 2)) \ 3$

$\langle \text{proof} \rangle$

**lemma** *beta-3*:

```

assumes  $ii < n$ 
shows blocky  $(\beta \ u) (\gamma \ (2 * ii + 1)) \ 2$ 

```

$\langle \text{proof} \rangle$

**lemma beta-4:**

**assumes**  $ii < p \ n$  **and**  $length \ u = p \ n$   
**shows**  $blocky \ (\beta \ u) \ (\gamma \ (2*n+2+2*ii+1)) \ 2$   
*<proof>*

**lemma beta-5:**

**assumes**  $ii < T'$   
**shows**  $blocky \ (\beta \ u) \ (\gamma \ (2*n+2*p \ n + 3 + ii)) \ 0$   
*<proof>*

**lemma beta-6:**

**assumes**  $ii < n$   
**shows**  $blocky \ (\beta \ u) \ (\gamma \ (2 * ii + 2)) \ (if \ x \ ! \ ii \ then \ 3 \ else \ 2)$   
*<proof>*

**lemma beta-7:**

**assumes**  $ii < p \ n$  **and**  $length \ u = p \ n$   
**shows**  $blocky \ (\beta \ u) \ (\gamma \ (2 * n + 4 + 2 * ii)) \ (if \ u \ ! \ ii \ then \ 3 \ else \ 2)$   
*<proof>*

**lemma beta-zeta0:**

**assumes**  $t \leq T'$   
**shows**  $blocky \ (\beta \ u) \ (\zeta_0 \ t) \ (z0 \ u \ t)$   
*<proof>*

**lemma beta-zeta1:**

**assumes**  $t \leq T'$   
**shows**  $blocky \ (\beta \ u) \ (\zeta_1 \ t) \ (z1 \ u \ t)$   
*<proof>*

**lemma beta-zeta2:**

**assumes**  $t \leq T'$   
**shows**  $blocky \ (\beta \ u) \ (\zeta_2 \ t) \ (z2 \ u \ t)$   
*<proof>*

We can finally show that  $\beta(u)$  satisfies  $\Phi$  if  $u$  is a certificate for  $x$ .

**lemma satisfies-beta-PHI:**

**assumes**  $length \ u = p \ n$  **and**  $exc \ \langle x; u \rangle \ T' \ \langle . \rangle \ 1 = 1$   
**shows**  $\beta \ u \models \Phi$   
*<proof>*

**corollary ex-u-imp-sat-PHI:**

**assumes**  $length \ u = p \ n$  **and**  $exc \ \langle x; u \rangle \ T' \ \langle . \rangle \ 1 = 1$   
**shows**  $satisfiable \ \Phi$   
*<proof>*

The formula  $\Phi$  has the desired property:

**theorem L-iff-satisfiable:**  $x \in L \iff satisfiable \ \Phi$   
*<proof>*

**end**

**end**

## Chapter 7

# Auxiliary Turing machines for reducing $\mathcal{NP}$ languages to SAT

```
theory Aux-TM-Reducing
  imports Reducing
begin
```

In the previous chapter we have seen how to reduce a language  $L \in \mathcal{NP}$  to SAT by constructing for every string  $x$  a CNF formula  $\Phi$  that is satisfiable iff.  $x \in L$ . To complete the Cook-Levin theorem it remains to show that there is a polynomial-time Turing machine that on input  $x$  outputs  $\Phi$ . Constructing such a TM will be the subject of the rest of this article and conclude in the next chapter. This chapter introduces several TMs used in the construction.

### 7.1 Generating literals

Our representation of CNF formulas as lists of lists of numbers is based on a representation of literals as numbers. Our function *literal-n* encodes the positive literal  $v_i$  as the number  $2i + 1$  and the negative literal  $\bar{v}_i$  as  $2i$ . We already have the Turing machine *tm-times2* to cover the second case. Now we build a TM for the first case, that is, for doubling and incrementing.

```
definition tm-times2incr :: tapeidx  $\Rightarrow$  machine where
  tm-times2incr  $j \equiv$  tm-times2  $j$  ;; tm-incr  $j$ 
```

```
lemma tm-times2incr-tm:
  assumes  $0 < j$  and  $j < k$  and  $G \geq 4$ 
  shows turing-machine  $k$   $G$  (tm-times2incr  $j$ )
   $\langle$ proof $\rangle$ 
```

```
lemma transforms-tm-times2incrI [transforms-intros]:
  fixes  $j$  :: tapeidx
  fixes  $k$  :: nat and  $tps$   $tps'$  :: tape list
  assumes  $k \geq 2$  and  $j > 0$  and  $j < k$  and  $length$   $tps = k$ 
  assumes  $tps ! j = (\lfloor n \rfloor_N, 1)$ 
  assumes  $t = 12 + 4 * nlength$   $n$ 
  assumes  $tps' = tps[j := (\lfloor Suc (2 * n) \rfloor_N, 1)]$ 
  shows transforms (tm-times2incr  $j$ )  $tps$   $t$   $tps'$ 
   $\langle$ proof $\rangle$ 
```

```
lemma literal-n-rename:
  assumes  $v \div 2 < length$   $\sigma$ 
  shows  $2 * \sigma ! (v \div 2) + v \mod 2 = (literal-n \circ rename$   $\sigma) (n-literal$   $v)$ 
   $\langle$ proof $\rangle$ 
```

Combining *tm-times2* and *tm-times2incr*, the next Turing machine accepts a variable index  $i$  on tape  $j_1$  and a flag  $b$  on tape  $j_2$  and outputs on tape  $j_1$  the encoding of the positive literal  $v_i$  or the negative literal  $\bar{v}_i$  if  $b$  is positive or zero, respectively.

**definition** *tm-to-literal* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-to-literal j1 j2  $\equiv$ 
  IF  $\lambda rs. rs ! j2 = \square$  THEN
    tm-times2 j1
  ELSE
    tm-times2incr j1
  ENDIF

```

**lemma** *tm-to-literal-tm*:

```

assumes  $k \geq 2$  and  $G \geq 4$  and  $0 < j1$  and  $j1 < k$  and  $j2 < k$ 
shows turing-machine  $k$   $G$  (tm-to-literal  $j1$   $j2$ )
<proof>

```

**lemma** *transforms-tm-to-literalI* [*transforms-intros*]:

```

fixes  $j1$   $j2$  :: tapeidx
fixes  $tps$   $tps'$  :: tape list and  $t$   $k$   $i$   $b$  :: nat
assumes  $0 < j1$   $j1 < k$   $j2 < k$   $2 \leq k$  length  $tps = k$ 
assumes
   $tps ! j1 = (\lfloor i \rfloor_N, 1)$ 
   $tps ! j2 = (\lfloor b \rfloor_N, 1)$ 
assumes  $t = 13 + 4 * nlength\ i$ 
assumes  $tps' = tps$ 
   $[j1 := (\lfloor 2 * i + (if\ b = 0\ then\ 0\ else\ 1) \rfloor_N, 1)]$ 
shows transforms (tm-to-literal  $j1$   $j2$ )  $tps$   $t$   $tps'$ 
<proof>

```

## 7.2 A Turing machine for relabeling formulas

In order to construct  $\Phi_9$ , we must construct CNF formulas  $\chi_t$ , which have the form  $\varrho(\psi)$  or  $\varrho'(\psi')$ . So we need a Turing machine for relabeling formulas. In this section we devise a Turing machine that gets a substitution  $\sigma$  and a CNF formula  $\varphi$  and outputs  $\sigma(\varphi)$ . In order to bound its running time we first prove some bounds on the length of relabeled formulas.

### 7.2.1 The length of relabeled formulas

First we bound the length of the representation of a single relabeled clause. In the following lemma the assumption ensures that the substitution  $\sigma$  has a value for every variable in the clause.

**lemma** *nlength-rename*:

```

assumes  $\forall v \in set\ clause. v\ div\ 2 < length\ \sigma$ 
shows  $nlength\ (map\ (literal\text{-}n\ \circ\ rename\ \sigma)\ (n\text{-}clause\ clause)) \leq length\ clause * Suc\ (nlength\ \sigma)$ 
<proof>

```

Our upper bound for the length of the symbol representation of a relabeled formula is fairly crude. It is basically the length of the string resulting from replacing every symbol of the original formula by the entire substitution.

**lemma** *nllength-relabel*:

```

assumes  $\forall clause \in set\ \varphi. \forall v \in set\ (clause\text{-}n\ clause). v\ div\ 2 < length\ \sigma$ 
shows  $nllength\ (formula\text{-}n\ (relabel\ \sigma\ \varphi)) \leq Suc\ (nlength\ \sigma) * nllength\ (formula\text{-}n\ \varphi)$ 
<proof>

```

A simple sufficient condition for the assumption in the previous lemma.

**lemma** *variables- $\sigma$* :

```

assumes  $variables\ \varphi \subseteq \{..<length\ \sigma\}$ 
shows  $\forall clause \in set\ \varphi. \forall v \in set\ (clause\text{-}n\ clause). v\ div\ 2 < length\ \sigma$ 
<proof>

```

Combining the previous two lemmas yields this upper bound:

**lemma** *nllength-relabel-variables*:

```

assumes  $variables\ \varphi \subseteq \{..<length\ \sigma\}$ 
shows  $nllength\ (formula\text{-}n\ (relabel\ \sigma\ \varphi)) \leq Suc\ (nlength\ \sigma) * nllength\ (formula\text{-}n\ \varphi)$ 
<proof>

```

## 7.2.2 Relabeling clauses

Relabeling a CNF formula is accomplished by relabeling each of its clauses. In this section we devise a Turing machine for relabeling clauses. The TM accepts on tape  $j$  a list of numbers representing a substitution  $\sigma$  and on tape  $j + 1$  a clause represented as a list of numbers. It outputs on tape  $j + 2$  the relabeled clause, consuming the original clause on tape  $j + 1$  in the process.

**definition** *tm-relabel-clause* :: *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-relabel-clause j  $\equiv$ 
  WHILE [] ;  $\lambda$ rs. rs ! (j + 1)  $\neq$   $\square$  DO
    tm-nextract | (j + 1) (j + 3) ;;
    tm-mod2 (j + 3) (j + 4) ;;
    tm-div2 (j + 3) ;;
    tm-nth-inplace j (j + 3) | ;;
    tm-to-literal (j + 3) (j + 4) ;;
    tm-append (j + 2) (j + 3) ;;
    tm-setn (j + 3) 0 ;;
    tm-setn (j + 4) 0
  DONE ;;
  tm-cr (j + 2) ;;
  tm-erase-cr (j + 1)

```

**lemma** *tm-relabel-clause-tm*:

```

assumes G  $\geq$  5 and j + 4 < k and 0 < j
shows turing-machine k G (tm-relabel-clause j)
<proof>

```

**locale** *turing-machine-relabel-clause* =

```

fixes j :: tapeidx
begin

```

```

definition tm1  $\equiv$  tm-nextract | (j + 1) (j + 3)
definition tm2  $\equiv$  tm1 ;; tm-mod2 (j + 3) (j + 4)
definition tm3  $\equiv$  tm2 ;; tm-div2 (j + 3)
definition tm4  $\equiv$  tm3 ;; tm-nth-inplace j (j + 3) |
definition tm5  $\equiv$  tm4 ;; tm-to-literal (j + 3) (j + 4)
definition tm6  $\equiv$  tm5 ;; tm-append (j + 2) (j + 3)
definition tm7 = tm6 ;; tm-setn (j + 3) 0
definition tm8  $\equiv$  tm7 ;; tm-setn (j + 4) 0
definition tmL  $\equiv$  WHILE [] ;  $\lambda$ rs. rs ! (j + 1)  $\neq$   $\square$  DO tm8 DONE
definition tm9  $\equiv$  tmL ;; tm-cr (j + 2)
definition tm10  $\equiv$  tm9 ;; tm-erase-cr (j + 1)

```

**lemma** *tm10-eq-tm-relabel-clause*: tm10 = tm-relabel-clause j

<proof>

**context**

```

fixes tps0 :: tape list and k :: nat and  $\sigma$  clause :: nat list
assumes clause:  $\forall v \in \text{set clause. } v \text{ div } 2 < \text{length } \sigma$ 
assumes jk: 0 < j j + 4 < k length tps0 = k
assumes tps0:

```

```

  tps0 ! j = ( $\lfloor \sigma \rfloor_{NL}$ , 1)
  tps0 ! (j + 1) = ( $\lfloor \text{clause} \rfloor_{NL}$ , 1)
  tps0 ! (j + 2) = ( $\lfloor [] \rfloor_{NL}$ , 1)
  tps0 ! (j + 3) = ( $\lfloor 0 \rfloor_N$ , 1)
  tps0 ! (j + 4) = ( $\lfloor 0 \rfloor_N$ , 1)

```

**begin**

**definition** tpsL t  $\equiv$  tps0

```

  [j + 1 := nltape' clause t,
   j + 2 := nltape (take t (map literal-n (map (rename  $\sigma$ ) (n-clause clause))))]

```

**lemma** *tpsL-eq-tps0*: tpsL 0 = tps0

<proof>

**definition**  $tps1\ t \equiv tps0$

$[j + 1 := nltape' \text{ clause } (Suc\ t),$   
 $j + 2 := nltape \text{ (take } t \text{ (map literal-} n \text{ (map (rename } \sigma \text{) (n-clause clause))))},$   
 $j + 3 := (\lfloor \text{clause ! } t \rfloor_N, 1)]$

**lemma**  $tm1$  [transforms-intros]:

**assumes**  $t < \text{length clause}$   
**and**  $ttt = 12 + 2 * \text{nlength (clause ! } t)$   
**shows**  $\text{transforms } tm1 \text{ (tpsL } t) \text{ ttt (tps1 } t)$   
 $\langle \text{proof} \rangle$

**definition**  $tps2\ t \equiv tps0$

$[j + 1 := nltape' \text{ clause } (Suc\ t),$   
 $j + 2 := nltape \text{ (take } t \text{ (map literal-} n \text{ (map (rename } \sigma \text{) (n-clause clause))))},$   
 $j + 3 := (\lfloor \text{clause ! } t \rfloor_N, 1),$   
 $j + 4 := (\lfloor (\text{clause ! } t) \bmod 2 \rfloor_N, 1)]$

**lemma**  $tm2$  [transforms-intros]:

**assumes**  $t < \text{length clause}$   
**and**  $ttt = 13 + 2 * \text{nlength (clause ! } t)$   
**shows**  $\text{transforms } tm2 \text{ (tpsL } t) \text{ ttt (tps2 } t)$   
 $\langle \text{proof} \rangle$

**definition**  $tps3\ t \equiv tps0$

$[j + 1 := nltape' \text{ clause } (Suc\ t),$   
 $j + 2 := nltape \text{ (take } t \text{ (map literal-} n \text{ (map (rename } \sigma \text{) (n-clause clause))))},$   
 $j + 3 := (\lfloor \text{clause ! } t \text{ div } 2 \rfloor_N, 1),$   
 $j + 4 := (\lfloor \text{clause ! } t \bmod 2 \rfloor_N, 1)]$

**lemma**  $tm3$  [transforms-intros]:

**assumes**  $t < \text{length clause}$   
**and**  $ttt = 16 + 4 * \text{nlength (clause ! } t)$   
**shows**  $\text{transforms } tm3 \text{ (tpsL } t) \text{ ttt (tps3 } t)$   
 $\langle \text{proof} \rangle$

**definition**  $tps4\ t \equiv tps0$

$[j + 1 := nltape' \text{ clause } (Suc\ t),$   
 $j + 2 := nltape \text{ (take } t \text{ (map literal-} n \text{ (map (rename } \sigma \text{) (n-clause clause))))},$   
 $j + 3 := (\lfloor \sigma ! (\text{clause ! } t \text{ div } 2) \rfloor_N, 1),$   
 $j + 4 := (\lfloor \text{clause ! } t \bmod 2 \rfloor_N, 1)]$

**lemma**  $tm4$  [transforms-intros]:

**assumes**  $t < \text{length clause}$   
**and**  $ttt = 28 + 4 * \text{nlength (clause ! } t) + 18 * (\text{nlength } \sigma)^2$   
**shows**  $\text{transforms } tm4 \text{ (tpsL } t) \text{ ttt (tps4 } t)$   
 $\langle \text{proof} \rangle$

**definition**  $tps5\ t \equiv tps0$

$[j + 1 := nltape' \text{ clause } (Suc\ t),$   
 $j + 2 := nltape \text{ (take } t \text{ (map literal-} n \text{ (map (rename } \sigma \text{) (n-clause clause))))},$   
 $j + 3 := (\lfloor 2 * (\sigma ! (\text{clause ! } t \text{ div } 2)) + \text{clause ! } t \bmod 2 \rfloor_N, 1),$   
 $j + 4 := (\lfloor \text{clause ! } t \bmod 2 \rfloor_N, 1)]$

**lemma**  $tm5$  [transforms-intros]:

**assumes**  $t < \text{length clause}$   
**and**  $ttt = 41 + 4 * \text{nlength (clause ! } t) + 18 * (\text{nlength } \sigma)^2 +$   
 $4 * \text{nlength } (\sigma ! (\text{clause ! } t \text{ div } 2))$   
**shows**  $\text{transforms } tm5 \text{ (tpsL } t) \text{ ttt (tps5 } t)$   
 $\langle \text{proof} \rangle$

**definition**  $tps6\ t \equiv tps0$

$[j + 1 := nltape' \text{ clause } (Suc\ t),$

$j + 2 := \text{nltape } (\text{take } (\text{Suc } t) (\text{map literal-}n (\text{map } (\text{rename } \sigma) (n\text{-clause clause}))))),$   
 $j + 3 := ([2 * (\sigma ! (\text{clause} ! t \text{ div } 2)) + \text{clause} ! t \text{ mod } 2]_N, 1),$   
 $j + 4 := ([\text{clause} ! t \text{ mod } 2]_N, 1)]$

**lemma tm6:**

**assumes**  $t < \text{length clause}$   
**and**  $\text{tnt} = 47 + 4 * \text{nlength } (\text{clause} ! t) + 18 * (\text{nlength } \sigma)^2 +$   
 $4 * \text{nlength } (\sigma ! (\text{clause} ! t \text{ div } 2)) +$   
 $2 * \text{nlength } (2 * \sigma ! (\text{clause} ! t \text{ div } 2) + \text{clause} ! t \text{ mod } 2)$   
**shows** *transforms tm6 (tpsL t) tnt (tps6 t)*  
 $\langle \text{proof} \rangle$

**lemma nlength- $\sigma$ 1:**

**assumes**  $t < \text{length clause}$   
**shows**  $\text{nlength } (\text{clause} ! t) \leq \text{nlength } \sigma$   
 $\langle \text{proof} \rangle$

**lemma nlength- $\sigma$ 2:**

**assumes**  $t < \text{length clause}$   
**shows**  $\text{nlength } (\sigma ! (\text{clause} ! t \text{ div } 2)) \leq \text{nlength } \sigma$   
 $\langle \text{proof} \rangle$

**lemma nlength- $\sigma$ 3:**

**assumes**  $t < \text{length clause}$   
**shows**  $\text{nlength } (2 * \sigma ! (\text{clause} ! t \text{ div } 2) + \text{clause} ! t \text{ mod } 2) \leq \text{Suc } (\text{nlength } \sigma)$   
 $\langle \text{proof} \rangle$

**lemma tm6' [transforms-intros]:**

**assumes**  $t < \text{length clause}$   
**and**  $\text{tnt} = 49 + 28 * \text{nlength } \sigma \wedge 2$   
**shows** *transforms tm6 (tpsL t) tnt (tps6 t)*  
 $\langle \text{proof} \rangle$

**definition tps7 t  $\equiv$  tps0**

$[j + 1 := \text{nltape}' \text{ clause } (\text{Suc } t),$   
 $j + 2 := \text{nltape } (\text{take } (\text{Suc } t) (\text{map literal-}n (\text{map } (\text{rename } \sigma) (n\text{-clause clause}))))),$   
 $j + 4 := ([\text{clause} ! t \text{ mod } 2]_N, 1)]$

**lemma tm7:**

**assumes**  $t < \text{length clause}$   
**and**  $\text{tnt} = 59 + 28 * \text{nlength } \sigma \wedge 2 +$   
 $2 * \text{nlength } (2 * \sigma ! (\text{clause} ! t \text{ div } 2) + \text{clause} ! t \text{ mod } 2)$   
**shows** *transforms tm7 (tpsL t) tnt (tps7 t)*  
 $\langle \text{proof} \rangle$

**lemma tm7' [transforms-intros]:**

**assumes**  $t < \text{length clause}$   
**and**  $\text{tnt} = 61 + 30 * \text{nlength } \sigma \wedge 2$   
**shows** *transforms tm7 (tpsL t) tnt (tps7 t)*  
 $\langle \text{proof} \rangle$

**definition tps8 t  $\equiv$  tps0**

$[j + 1 := \text{nltape}' \text{ clause } (\text{Suc } t),$   
 $j + 2 := \text{nltape } (\text{take } (\text{Suc } t) (\text{map literal-}n (\text{map } (\text{rename } \sigma) (n\text{-clause clause}))))]$

**lemma tm8:**

**assumes**  $t < \text{length clause}$   
**and**  $\text{tnt} = 61 + 30 * (\text{nlength } \sigma)^2 + (10 + 2 * \text{nlength } (\text{clause} ! t \text{ mod } 2))$   
**shows** *transforms tm8 (tpsL t) tnt (tps8 t)*  
 $\langle \text{proof} \rangle$

**lemma tm8' [transforms-intros]:**

**assumes**  $t < \text{length clause}$

**and**  $t_{tt} = 71 + 32 * (\text{nlength } \sigma)^2$   
**shows** *transforms tm8* (*tpsL t*) *t\_{tt}* (*tpsL (Suc t)*)  
*<proof>*

**lemma** *tmL* [*transforms-intros*]:  
**assumes**  $t_{tt} = \text{length clause} * (73 + 32 * (\text{nlength } \sigma)^2) + 1$   
**shows** *transforms tmL* (*tpsL 0*) *t\_{tt}* (*tpsL (length clause)*)  
*<proof>*

**lemma** *tpsL-length*: *tpsL (length clause) = tps0*  
 $[j + 1 := \text{nltape}' \text{ clause } (\text{length clause}),$   
 $j + 2 := \text{nltape} (\text{map literal-n } (\text{map } (\text{rename } \sigma) (\text{n-clause clause})))]$   
*<proof>*

**definition** *tps9*  $\equiv$  *tps0*  
 $[j + 1 := \text{nltape}' \text{ clause } (\text{length clause}),$   
 $j + 2 := (\lfloor \text{map literal-n } (\text{map } (\text{rename } \sigma) (\text{n-clause clause})) \rfloor_{NL}, 1)]$

**lemma** *tm9*:  
**assumes**  $t_{tt} = 4 + \text{length clause} * (73 + 32 * (\text{nlength } \sigma)^2) +$   
 $\text{nlength } (\text{map } (\text{literal-n } \circ \text{rename } \sigma) (\text{n-clause clause}))$   
**shows** *transforms tm9* (*tpsL 0*) *t\_{tt}* *tps9*  
*<proof>*

**lemma** *tm9'* [*transforms-intros*]:  
**assumes**  $t_{tt} = 4 + 2 * \text{length clause} * (73 + 32 * (\text{nlength } \sigma)^2)$   
**shows** *transforms tm9* *tps0* *t\_{tt}* *tps9*  
*<proof>*

**definition** *tps10*  $\equiv$  *tps0*  
 $[j + 1 := (\lfloor \rfloor_{NL}, 1),$   
 $j + 2 := (\lfloor \text{map literal-n } (\text{map } (\text{rename } \sigma) (\text{n-clause clause})) \rfloor_{NL}, 1)]$

**lemma** *tm10*:  
**assumes**  $t_{tt} = 11 + 2 * \text{length clause} * (73 + 32 * (\text{nlength } \sigma)^2) + 3 * \text{nlength clause}$   
**shows** *transforms tm10* *tps0* *t\_{tt}* *tps10*  
*<proof>*

**lemma** *tm10'*:  
**assumes**  $t_{tt} = 11 + 64 * \text{nlength clause} * (3 + (\text{nlength } \sigma)^2)$   
**shows** *transforms tm10* *tps0* *t\_{tt}* *tps10*  
*<proof>*

**end**

**end**

**lemma** *transforms-tm-relabel-clauseI* [*transforms-intros*]:  
**fixes**  $j :: \text{tapeidx}$   
**fixes**  $\text{tps } \text{tps}' :: \text{tape list}$  **and**  $t_{tt} \ k :: \text{nat}$  **and**  $\sigma \ \text{clause} :: \text{nat list}$   
**assumes**  $0 < j$   $j + 4 < k$   $\text{length } \text{tps} = k$   
**and**  $\forall v \in \text{set clause. } v \ \text{div } 2 < \text{length } \sigma$   
**assumes**  
 $\text{tps} ! j = (\lfloor \sigma \rfloor_{NL}, 1)$   
 $\text{tps} ! (j + 1) = (\lfloor \text{clause} \rfloor_{NL}, 1)$   
 $\text{tps} ! (j + 2) = (\lfloor \rfloor_{NL}, 1)$   
 $\text{tps} ! (j + 3) = (\lfloor 0 \rfloor_N, 1)$   
 $\text{tps} ! (j + 4) = (\lfloor 0 \rfloor_N, 1)$   
**assumes**  $t_{tt} = 11 + 64 * \text{nlength clause} * (3 + (\text{nlength } \sigma)^2)$   
**assumes**  $\text{tps}' = \text{tps}$   
 $[j + 1 := (\lfloor \rfloor_{NL}, 1),$   
 $j + 2 := (\lfloor \text{clause-n } (\text{map } (\text{rename } \sigma) (\text{n-clause clause})) \rfloor_{NL}, 1)]$   
**shows** *transforms (tm-relabel-clause j)* *tps* *t\_{tt}* *tps'*

*<proof>*

### 7.2.3 Relabeling CNF formulas

A Turing machine can relabel a CNF formula by relabeling each clause using the TM *tm-relabel-clause*. The next TM accepts a CNF formula as a list of lists of literals on tape  $j_1$  and a substitution  $\sigma$  as a list of numbers on tape  $j_2 + 1$ . It outputs the relabeled formula on tape  $j_2$ , which initially must be empty.

**definition** *tm-relabel* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

```
tm-relabel j1 j2  $\equiv$ 
  WHILE [] ;  $\lambda$ rs. rs ! j1  $\neq$  [] DO
    tm-nextract # j1 (j2 + 2) ;;
    tm-relabel-clause (j2 + 1) ;;
    tm-appendl j2 (j2 + 3) ;;
    tm-erase-cr (j2 + 3)
  DONE ;;
  tm-cr j1 ;;
  tm-cr j2
```

**lemma** *tm-relabel-tm*:

```
assumes  $G \geq 6$  and  $j_2 + 5 < k$  and  $0 < j_1$  and  $j_1 < j_2$ 
shows turing-machine  $k$   $G$  (tm-relabel j1 j2)
<proof>
```

**locale** *turing-machine-relabel* =

```
fixes j1 j2 :: tapeidx
```

**begin**

**definition** *tmL1*  $\equiv$  *tm-nextract* # *j1* (*j2* + 2)

**definition** *tmL2*  $\equiv$  *tmL1* ;; *tm-relabel-clause* (*j2* + 1)

**definition** *tmL3*  $\equiv$  *tmL2* ;; *tm-appendl* *j2* (*j2* + 3)

**definition** *tmL4*  $\equiv$  *tmL3* ;; *tm-erase-cr* (*j2* + 3)

**definition** *tmL*  $\equiv$  WHILE [] ;  $\lambda$ rs. rs ! *j1*  $\neq$  [] DO *tmL4* DONE

**definition** *tm2*  $\equiv$  *tmL* ;; *tm-cr* *j1*

**definition** *tm3*  $\equiv$  *tm2* ;; *tm-cr* *j2*

**lemma** *tm3-eq-tm-relabel*: *tm3* = *tm-relabel* *j1 j2*

*<proof>*

**context**

```
fixes tps0 :: tape list and  $k$  :: nat and  $\sigma$  :: nat list and  $\varphi$  :: formula
```

```
assumes variables: variables  $\varphi \subseteq \{..<length\ \sigma\}$ 
```

```
assumes jk:  $0 < j_1$   $j_1 < j_2$   $j_2 + 5 < k$  length tps0 =  $k$ 
```

```
assumes tps0:
```

```
  tps0 ! j1 = ([formula-n  $\varphi$ ]NLL, 1)
```

```
  tps0 ! j2 = ([ ]NLL, 1)
```

```
  tps0 ! (j2 + 1) = ([ $\sigma$ ]NL, 1)
```

```
  tps0 ! (j2 + 2) = ([ ]NL, 1)
```

```
  tps0 ! (j2 + 3) = ([ ]NL, 1)
```

```
  tps0 ! (j2 + 4) = ([0]N, 1)
```

```
  tps0 ! (j2 + 5) = ([0]N, 1)
```

**begin**

**abbreviation** *n $\varphi$*  :: *nat list list* **where**

```
n $\varphi$   $\equiv$  formula-n  $\varphi$ 
```

**definition** *tpsL* :: *nat*  $\Rightarrow$  *tape list* **where**

```
tpsL t  $\equiv$  tps0
```

```
  [j1 := nlltape' n $\varphi$  t,
```

```
  j2 := nlltape (formula-n (take t (relabel  $\sigma$   $\varphi$ )))]
```

**lemma** *tpsL-eq-tps0*: *tpsL* 0 = *tps0*

*<proof>*

**definition**  $tpsL1 :: nat \Rightarrow tape\ list\ where$

$tpsL1\ t \equiv tps0$   
 $[j1 := nlltape'\ n\varphi\ (Suc\ t),$   
 $j2 := nlltape\ (formula\text{-}n\ (take\ t\ (relabel\ \sigma\ \varphi))),$   
 $j2 + 2 := (\lfloor n\varphi ! t \rfloor_{NL}, 1)]$

**lemma**  $tmL1$  [transforms-intros]:

**assumes**  $ttt = 12 + 2 * nllength\ (n\varphi ! t)$   
**and**  $t < length\ \varphi$   
**shows**  $transforms\ tmL1\ (tpsL\ t)\ ttt\ (tpsL1\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL2 :: nat \Rightarrow tape\ list\ where$

$tpsL2\ t \equiv tps0$   
 $[j1 := nlltape'\ n\varphi\ (Suc\ t),$   
 $j2 := nlltape\ (formula\text{-}n\ (take\ t\ (relabel\ \sigma\ \varphi))),$   
 $j2 + 2 := (\lfloor \rfloor_{NL}, 1),$   
 $j2 + 3 := (\lfloor clause\text{-}n\ (map\ (rename\ \sigma)\ (n\text{-}clause\ (n\varphi ! t))) \rfloor_{NL}, 1)]$

**lemma**  $tmL2$  [transforms-intros]:

**assumes**  $ttt = 23 + 2 * nllength\ (n\varphi ! t) + 64 * nllength\ (n\varphi ! t) * (3 + (nllength\ \sigma)^2)$   
**and**  $t < length\ \varphi$   
**shows**  $transforms\ tmL2\ (tpsL\ t)\ ttt\ (tpsL2\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL3 :: nat \Rightarrow tape\ list\ where$

$tpsL3\ t \equiv tps0$   
 $[j1 := nlltape'\ n\varphi\ (Suc\ t),$   
 $j2 := nlltape$   
 $(formula\text{-}n\ (take\ t\ (relabel\ \sigma\ \varphi))\ @\ [clause\text{-}n\ (map\ (rename\ \sigma)\ (n\text{-}clause\ (n\varphi ! t))]),$   
 $j2 + 2 := (\lfloor \rfloor_{NL}, 1),$   
 $j2 + 3 := (\lfloor clause\text{-}n\ (map\ (rename\ \sigma)\ (n\text{-}clause\ (n\varphi ! t))) \rfloor_{NL}, 1)]$

**lemma**  $tmL3$  [transforms-intros]:

**assumes**  $ttt = 29 + 2 * nllength\ (n\varphi ! t) +$   
 $64 * nllength\ (n\varphi ! t) * (3 + (nllength\ \sigma)^2) +$   
 $2 * nllength\ (clause\text{-}n\ (map\ (rename\ \sigma)\ (n\text{-}clause\ (n\varphi ! t))))$   
**and**  $t < length\ \varphi$   
**shows**  $transforms\ tmL3\ (tpsL\ t)\ ttt\ (tpsL3\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL4 :: nat \Rightarrow tape\ list\ where$

$tpsL4\ t \equiv tps0$   
 $[j1 := nlltape'\ n\varphi\ (Suc\ t),$   
 $j2 := nlltape$   
 $(formula\text{-}n\ (take\ t\ (relabel\ \sigma\ \varphi))\ @\ [clause\text{-}n\ (map\ (rename\ \sigma)\ (n\text{-}clause\ (n\varphi ! t))]),$   
 $j2 + 2 := (\lfloor \rfloor_{NL}, 1)]$

**lemma**  $tmL4$ :

**assumes**  $ttt = 36 + 2 * nllength\ (n\varphi ! t) +$   
 $64 * nllength\ (n\varphi ! t) * (3 + (nllength\ \sigma)^2) +$   
 $4 * nllength\ (clause\text{-}n\ (map\ (rename\ \sigma)\ (n\text{-}clause\ (n\varphi ! t))))$   
**and**  $t < length\ \varphi$   
**shows**  $transforms\ tmL4\ (tpsL\ t)\ ttt\ (tpsL4\ t)$   
 $\langle proof \rangle$

**lemma**  $nllength\text{-}1$ :

**assumes**  $t < length\ \varphi$   
**shows**  $nllength\ (n\varphi ! t) \leq nllength\ n\varphi$   
 $\langle proof \rangle$

**lemma**  $nllength\text{-}2$ :

**assumes**  $t < length\ \varphi$

**shows**  $nllength (clause\text{-}n (map (rename \sigma) (n\text{-}clause (n\varphi ! t)))) \leq nllength (formula\text{-}n (relabel \sigma \varphi))$   
**(is ?l ≤ ?r)**  
 ⟨proof⟩

**definition**  $tpsL4' t \equiv tps0$   
 $[j1 := nlltape' n\varphi (Suc t),$   
 $j2 := nlltape (formula\text{-}n (take (Suc t) (relabel \sigma \varphi)))]$

**lemma**  $tpsL4$ :  
**assumes**  $t < length \varphi$   
**shows**  $tpsL4 t = tpsL4' t$   
 ⟨proof⟩

**lemma**  $tpsL4'\text{-}eq\text{-}tpsL$ :  $tpsL4' t = tpsL (Suc t)$   
 ⟨proof⟩

**lemma**  $tmL4'$  [transforms-intros]:  
**assumes**  $ttt = 36 + 65 * nllength n\varphi * (3 + (nllength \sigma)^2) + 4 * nllength (formula\text{-}n (relabel \sigma \varphi))$   
**and**  $t < length \varphi$   
**shows** transforms  $tmL4 (tpsL t) ttt (tpsL (Suc t))$   
 ⟨proof⟩

**lemma**  $tmL$ :  
**assumes**  $ttt = length \varphi * (38 + 65 * nllength n\varphi * (3 + (nllength \sigma)^2) + 4 * nllength (formula\text{-}n (relabel \sigma \varphi))) + 1$   
**shows** transforms  $tmL (tpsL 0) ttt (tpsL (length \varphi))$   
 ⟨proof⟩

**lemma**  $tmL'$  [transforms-intros]:  
**assumes**  $ttt = 107 * nllength n\varphi ^ 2 * (3 + nllength \sigma ^ 2) + 1$   
**shows** transforms  $tmL (tpsL 0) ttt (tpsL (length \varphi))$   
 ⟨proof⟩

**definition**  $tps1 :: tape\ list\ where$   
 $tps1 \equiv tps0$   
 $[j1 := nlltape' n\varphi (length \varphi),$   
 $j2 := nlltape (formula\text{-}n (relabel \sigma \varphi))]$

**lemma**  $tps1\text{-}eq\text{-}tpsL$ :  $tps1 = tpsL (length \varphi)$   
 ⟨proof⟩

**definition**  $tps2 \equiv tps0$   
 $[j2 := nlltape (formula\text{-}n (relabel \sigma \varphi))]$

**lemma**  $tm2$  [transforms-intros]:  
**assumes**  $ttt = Suc (107 * (nllength n\varphi)^2 * (3 + (nllength \sigma)^2)) +$   
 $Suc (Suc (Suc (nllength n\varphi)))$   
**shows** transforms  $tm2 (tpsL 0) ttt tps2$   
 ⟨proof⟩

**definition**  $tps3 \equiv tps0$   
 $[j2 := nlltape' (formula\text{-}n (relabel \sigma \varphi)) 0]$

**lemma**  $tm3$ :  
**assumes**  $ttt = 7 + (107 * (nllength n\varphi)^2 * (3 + (nllength \sigma)^2)) +$   
 $nllength n\varphi + nllength (formula\text{-}n (relabel \sigma \varphi))$   
**shows** transforms  $tm3 (tpsL 0) ttt tps3$   
 ⟨proof⟩

**lemma**  $tm3'$  [transforms-intros]:  
**assumes**  $ttt = 7 + (108 * (nllength n\varphi)^2 * (3 + (nllength \sigma)^2))$   
**shows** transforms  $tm3 tps0 ttt tps3$   
 ⟨proof⟩

end

end

**lemma** *transforms-tm-relabelI* [*transforms-intros*]:

**fixes**  $j1\ j2 :: \text{tapeidx}$

**fixes**  $tps\ tps' :: \text{tape list}$  **and**  $t\ t\ k :: \text{nat}$  **and**  $\sigma :: \text{nat list}$  **and**  $\varphi :: \text{formula}$

**assumes**  $0 < j1$  **and**  $j1 < j2$  **and**  $j2 + 5 < k$  **and**  $\text{length } tps = k$

**and**  $\text{variables } \varphi \subseteq \{..<\text{length } \sigma\}$

**assumes**

$tps ! j1 = ([\text{formula-n } \varphi]_{NLL}, 1)$

$tps ! j2 = ([\ ]_{NLL}, 1)$

$tps ! (j2 + 1) = ([\sigma]_{NL}, 1)$

$tps ! (j2 + 2) = ([\ ]_{NL}, 1)$

$tps ! (j2 + 3) = ([\ ]_{NL}, 1)$

$tps ! (j2 + 4) = ([0]_N, 1)$

$tps ! (j2 + 5) = ([0]_N, 1)$

**assumes**  $tps' = tps$

$[j2 := \text{nlltape}' (\text{formula-n } (\text{relabel } \sigma \varphi))\ 0]$

**assumes**  $t\ t\ = 7 + (108 * (\text{nllength } (\text{formula-n } \varphi))^2 * (3 + (\text{nllength } \sigma)^2))$

**shows**  $\text{transforms } (\text{tm-relabel } j1\ j2)\ tps\ t\ t\ tps'$

*<proof>*

## 7.3 Listing the head positions of a Turing machine

The formulas  $\chi_t$  used for  $\Phi_9$  require the functions *inputpos* and *prev*, or more precisely the substitutions  $\varrho_t$  and  $\varrho'_t$  do.

In this section we build a TM that simulates a two-tape TM  $M$  on some input until it halts. During the simulation it creates two lists: one with the sequence of head positions on  $M$ 's input tape and one with the sequence of head positions on  $M$ 's output tape. The first list directly provides *inputpos*; the second list allows the computation of *prev* using the TM *tm-prev*.

### 7.3.1 Simulating and logging head movements

At the core of the simulation is the following Turing command. It interprets the tapes  $j + 7$  and  $j + 8$  as input tape and output tape of a two-tape Turing machine  $M$  and the symbol in the first cell of tape  $j + 6$  as the state of  $M$ . It then applies the actions of  $M$  in this configuration to the tapes  $j + 7$  and  $j + 8$  and adapts the state on tape  $j + 6$  accordingly. On top of that it writes (“logs”) to tape  $j$  the direction in which  $M$ 's input tape head has moved and to tape  $j + 3$  the direction in which  $M$ 's work tape head has moved.

The head movement directions are encoded by the symbols  $\square$ ,  $\triangleright$ , and  $\mathbf{0}$  for Left, Stay, and Right, respectively.

The command is parameterized by the TM  $M$  and its alphabet size  $G$  (and as usual the tape index  $j$ ). The command does nothing if the state on tape  $j + 6$  is the halting state or if the symbol read from the simulated tape  $j + 7$  or  $j + 8$  is outside the alphabet  $G$ .

**definition** *direction-to-symbol*  $:: \text{direction} \Rightarrow \text{symbol}$  **where**

*direction-to-symbol*  $d \equiv (\text{case } d \text{ of Left} \Rightarrow \square \mid \text{Stay} \Rightarrow \triangleright \mid \text{Right} \Rightarrow \mathbf{0})$

**lemma** *direction-to-symbol-less*: *direction-to-symbol*  $d < 3$

*<proof>*

**definition** *cmd-simulog*  $:: \text{nat} \Rightarrow \text{machine} \Rightarrow \text{tapeidx} \Rightarrow \text{command}$  **where**

*cmd-simulog*  $G\ M\ j\ rs \equiv$

(1,

*if*  $rs ! (j + 6) \geq \text{length } M \vee rs ! (j + 7) \geq G \vee rs ! (j + 8) \geq G$

*then*  $\text{map } (\lambda z. (z, \text{Stay}))\ rs$

*else*

```

map ( $\lambda i$ . let sas = (M ! (rs ! (j + 6))) [rs ! (j + 7), rs ! (j + 8)] in
  if i = j then (direction-to-symbol (sas [~] 0), Stay)
  else if i = j + 3 then (direction-to-symbol (sas [~] 1), Stay)
  else if i = j + 6 then (fst sas, Stay)
  else if i = j + 7 then sas [!] 0
  else if i = j + 8 then sas [!] 1
  else (rs ! i, Stay))
[0.. $\text{length } rs$ ]

```

**lemma** *turing-command-cmd-simulog*:

**fixes**  $G H :: \text{nat}$

**assumes** *turing-machine* 2  $G M$  **and**  $k \geq j + 9$  **and**  $j > 0$  **and**  $H \geq \text{Suc } (\text{length } M)$  **and**  $H \geq G$

**shows** *turing-command*  $k$  1  $H$  (*cmd-simulog*  $G M j$ )

*<proof>*

The logging Turing machine consists only of the logging command.

**definition** *tm-simulog*  $:: \text{nat} \Rightarrow \text{machine} \Rightarrow \text{tapeidx} \Rightarrow \text{machine}$  **where**

*tm-simulog*  $G M j \equiv [\text{cmd-simulog } G M j]$

**lemma** *tm-simulog-tm*:

**fixes**  $H :: \text{nat}$

**assumes** *turing-machine* 2  $G M$  **and**  $k \geq j + 9$  **and**  $j > 0$  **and**  $H \geq \text{Suc } (\text{length } M)$  **and**  $H \geq G$

**shows** *turing-machine*  $k$   $H$  (*tm-simulog*  $G M j$ )

*<proof>*

**lemma** *transforms-tm-simulogI* [*transforms-intros*]:

**fixes**  $G :: \text{nat}$  **and**  $M :: \text{machine}$  **and**  $j :: \text{tapeidx}$

**fixes**  $k :: \text{nat}$  **and**  $\text{tps } \text{tps}' :: \text{tape list}$  **and**  $\text{xs} :: \text{symbol list}$

**assumes** *turing-machine* 2  $G M$  **and**  $k \geq j + 9$  **and**  $j > 0$

**and** *symbols-lt*  $G \text{ xs}$

**and**  $\text{cfg} = \text{execute } M \text{ (start-config } 2 \text{ xs) } t$  **and**  $\text{fst } \text{cfg} < \text{length } M$

**and**  $\text{length } \text{tps} = k$

**assumes**

$\text{tps} ! j = [\text{dummy1}]$

$\text{tps} ! (j + 3) = [\text{dummy2}]$

$\text{tps} ! (j + 6) = [\text{fst } \text{cfg}]$

$\text{tps} ! (j + 7) = \text{cfg} <!\> 0$

$\text{tps} ! (j + 8) = \text{cfg} <!\> 1$

**assumes**  $\text{tps}' = \text{tps}$

$[j := [\text{direction-to-symbol } ((M ! (\text{fst } \text{cfg})) (\text{config-read } \text{cfg}) [\sim] 0)],$

$j + 3 := [\text{direction-to-symbol } ((M ! (\text{fst } \text{cfg})) (\text{config-read } \text{cfg}) [\sim] 1)],$

$j + 6 := [\text{fst } (\text{execute } M \text{ (start-config } 2 \text{ xs) } (\text{Suc } t))],$

$j + 7 := \text{execute } M \text{ (start-config } 2 \text{ xs) } (\text{Suc } t) <!\> 0,$

$j + 8 := \text{execute } M \text{ (start-config } 2 \text{ xs) } (\text{Suc } t) <!\> 1]$

**shows** *transforms* (*tm-simulog*  $G M j$ )  $\text{tps}$  1  $\text{tps}'$

*<proof>*

### 7.3.2 Adjusting head position counters

The Turing machine *tm-simulog* logs the head movements, but what we need is a list of all the head positions during the execution of  $M$ . The next TM maintains a number for a head position and adjusts it based on a head movement symbol as provided by *tm-simulog*.

More precisely, the next Turing machine accepts on tape  $j$  a symbol encoding a direction, on tape  $j + 1$  a number representing a head position, and on tape  $j + 2$  a list of numbers. Depending on the symbol on tape  $j$  it decreases, increases or leaves unchanged the number on tape  $j + 1$ . Then it appends this adjusted number to the list on tape  $j + 2$ .

**definition** *tm-adjust-headpos*  $:: \text{nat} \Rightarrow \text{machine}$  **where**

*tm-adjust-headpos*  $j \equiv$

*IF*  $\lambda rs$ .  $rs ! j = \square$  *THEN*

*tm-decr*  $(j + 1)$

*ELSE*

*IF*  $\lambda rs$ .  $rs ! j = \mathbf{0}$  *THEN*

```

    tm-incr (j + 1)
  ELSE
  []
  ENDIF
ENDIF ;;
tm-append (j + 2) (j + 1)

```

**lemma** *tm-adjust-headpos-tm*:  
**assumes**  $G \geq 5$  **and**  $j + 2 < k$   
**shows** *turing-machine*  $k$   $G$  (*tm-adjust-headpos*  $j$ )  
*<proof>*

**locale** *turing-machine-adjust-headpos* =  
**fixes**  $j :: \text{tapeid}$   
**begin**

**definition**  $tm1 \equiv \text{IF } \lambda rs. rs ! j = \mathbf{0} \text{ THEN } tm\text{-incr } (j + 1) \text{ ELSE } [] \text{ ENDIF}$   
**definition**  $tm2 \equiv \text{IF } \lambda rs. rs ! j = \square \text{ THEN } tm\text{-decr } (j + 1) \text{ ELSE } tm1 \text{ ENDIF}$   
**definition**  $tm3 \equiv tm2 ;; tm\text{-append } (j + 2) (j + 1)$

**lemma** *tm3-eq-tm-adjust-headpos*:  $tm3 = tm\text{-adjust-headpos } j$   
*<proof>*

**context**

**fixes**  $tps :: \text{tape list}$  **and**  $jj :: \text{tapeid}$  **and**  $k t :: \text{nat}$  **and**  $xs :: \text{symbol list}$   
**fixes**  $M :: \text{machine}$   
**fixes**  $G \text{ cfg}$   
**assumes**  $jk$ :  $\text{length } tps = k$   $k \geq j + 3$   $jj < 2$   
**assumes**  $M$ : *turing-machine*  $2$   $G$   $M$   
**assumes**  $xs$ : *symbols-lt*  $G$   $xs$   
**assumes**  $cfg$ : *execute*  $M$  (*start-config*  $2$   $xs$ )  $t$   $\text{fst } cfg < \text{length } M$   
**assumes**  $tps0$ :  
 $tps ! j = \lceil \text{direction-to-symbol } ((M ! (\text{fst } cfg)) (\text{config-read } cfg) [\sim] jj) \rceil$   
 $tps ! (j + 1) = (\lfloor \text{cfg } \langle \# \rangle jj \rfloor_N, 1)$   
 $tps ! (j + 2) = \text{nltape } (\text{map } (\lambda t. (\text{execute } M (\text{start-config } 2 \text{ } xs) t \langle \# \rangle jj))) [0..<Suc t]$

**begin**

**lemma** *k-ge-2*:  $2 \leq k$   
*<proof>*

**abbreviation**  $exc :: \text{symbol list} \Rightarrow \text{nat} \Rightarrow \text{config}$  **where**  
 $exc \ y \ tt \equiv \text{execute } M (\text{start-config } 2 \ y) \ tt$

**lemma** *read-tps-j*:  $\text{read } tps ! j = \text{direction-to-symbol } ((M ! (\text{fst } cfg)) (\text{config-read } cfg) [\sim] jj)$   
*<proof>*

**lemma** *write-symbol*:

$\exists v. \text{execute } M (\text{start-config } 2 \ xs) (Suc \ t) \langle ! \rangle jj = \text{act } (v, (M ! (\text{fst } cfg)) (\text{config-read } cfg) [\sim] jj) (cfg \langle ! \rangle jj)$   
*<proof>*

**lemma**  $tm1$  [*transforms-intros*]:

**assumes**  $ttt = 7 + 2 * \text{nlength } (cfg \langle \# \rangle jj)$   
**and**  $(M ! (\text{fst } cfg)) (\text{config-read } cfg) [\sim] jj \neq \text{Left}$   
**and**  $tps' = tps[j + 1 := (\lfloor \text{execute } M (\text{start-config } 2 \ xs) (Suc \ t) \langle \# \rangle jj \rfloor_N, 1)]$   
**shows** *transforms*  $tm1$   $tps$   $ttt$   $tps'$   
*<proof>*

**lemma**  $tm2$  [*transforms-intros*]:

**assumes**  $ttt = 10 + 2 * \text{nlength } (cfg \langle \# \rangle jj)$   
**and**  $tps' = tps[j + 1 := (\lfloor \text{execute } M (\text{start-config } 2 \ xs) (Suc \ t) \langle \# \rangle jj \rfloor_N, 1)]$   
**shows** *transforms*  $tm2$   $tps$   $ttt$   $tps'$   
*<proof>*

**lemma** *tm3*:  
**assumes**  $ttt = 16 + 2 * nlength\ (cfg\ \langle\#\rangle\ jj) + 2 * nlength\ (exc\ xs\ (Suc\ t)\ \langle\#\rangle\ jj)$   
**and**  $tps' = tps$   
 $[j + 1 := (\lfloor execute\ M\ (start\ config\ 2\ xs)\ (Suc\ t)\ \langle\#\rangle\ jj \rfloor_N, 1),$   
 $j + 2 := nltape\ (map\ (\lambda t. (execute\ M\ (start\ config\ 2\ xs)\ t\ \langle\#\rangle\ jj))\ [0..<Suc\ (Suc\ t)])]$   
**shows** *transforms tm3 tps ttt tps'*  
*<proof>*

**end**

**end**

**lemma** *transforms-tm-adjust-headposI* [*transforms-intros*]:  
**fixes**  $j :: tapeidx$   
**fixes**  $tps\ tps' :: tape\ list$  **and**  $k\ jj\ t :: nat$  **and**  $xs :: symbol\ list$   
**and**  $M :: machine$  **and**  $G :: nat$  **and**  $cfg :: config$   
**assumes** *turing-machine 2 G M*  
**and**  $length\ tps = k$  **and**  $k \geq j + 3$  **and**  $jj < 2$   
**and** *symbols-lt G xs*  
**and**  $cfg: cfg = execute\ M\ (start\ config\ 2\ xs)\ t\ fst\ cfg < length\ M$   
**assumes**  
 $tps\ !\ j = \lceil direction\ to\ symbol\ ((M\ !\ (fst\ cfg))\ (config\ read\ cfg)\ [\sim]\ jj) \rceil$   
 $tps\ !\ (j + 1) = (\lfloor cfg\ \langle\#\rangle\ jj \rfloor_N, 1)$   
 $tps\ !\ (j + 2) = nltape\ (map\ (\lambda t. (execute\ M\ (start\ config\ 2\ xs)\ t\ \langle\#\rangle\ jj))\ [0..<Suc\ t])$   
**assumes** *max-head-pos:  $\forall t. execute\ M\ (start\ config\ 2\ xs)\ t\ \langle\#\rangle\ jj \leq max\ head\ pos$*   
**assumes**  $ttt: ttt = 16 + 4 * nlength\ max\ head\ pos$   
**assumes**  $tps' = tps$   
 $[j + 1 := (\lfloor execute\ M\ (start\ config\ 2\ xs)\ (Suc\ t)\ \langle\#\rangle\ jj \rfloor_N, 1),$   
 $j + 2 := nltape\ (map\ (\lambda t. (execute\ M\ (start\ config\ 2\ xs)\ t\ \langle\#\rangle\ jj))\ [0..<Suc\ (Suc\ t)])]$   
**shows** *transforms (tm-adjust-headpos j) tps ttt tps'*  
*<proof>*

### 7.3.3 Listing the head positions

The next Turing machine is essentially a loop around the TM *tm-simulog*, which outputs head movements, combined with two instances of the TM *tm-adjust-headpos*, each of which maintains a head positions lists. The loop ends when the simulated machine reaches the halting state. If the simulated machine does not halt, neither does the simulator, but we will not consider this case when we analyze the semantics. The TM receives an input on tape  $j + 7$ . During the simulation of  $M$  this tape is a replica of the simulated machine's input tape, and tape  $j + 8$  is a replica of the work/output tape. The lists of the head positions will be on tapes  $j + 2$  and  $j + 5$  for the input tape and work/output tape, respectively.

**definition** *tm-list-headpos*  $:: nat \Rightarrow machine \Rightarrow tapeidx \Rightarrow machine$  **where**

```

tm-list-headpos G M j  $\equiv$ 
  tm-right-many {j + 1, j + 2, j + 4, j + 5} ;;
  tm-write (j + 6)  $\square$  ;;
  tm-append (j + 2) (j + 1) ;;
  tm-append (j + 5) (j + 4) ;;
  WHILE  $\square$  ;  $\lambda rs. rs\ !\ (j + 6) < length\ M$  DO
    tm-simulog G M j ;;
    tm-adjust-headpos j ;;
    tm-adjust-headpos (j + 3) ;;
    tm-write-many {j, j + 3}  $\triangleright$ 
  DONE ;;
  tm-write (j + 6)  $\triangleright$  ;;
  tm-cr (j + 2) ;;
  tm-cr (j + 5)
```

**lemma** *tm-list-headpos-tm*:  
**fixes**  $H :: nat$   
**assumes** *turing-machine 2 G M* **and**  $k \geq j + 9$  **and**  $j > 0$  **and**  $H \geq Suc\ (length\ M)$  **and**  $H \geq G$   
**assumes**  $H \geq 5$   
**shows** *turing-machine k H (tm-list-headpos G M j)*

*<proof>*

**locale** *turing-machine-list-headpos* =

**fixes**  $G :: \text{nat}$  **and**  $M :: \text{machine}$  **and**  $j :: \text{tapeidx}$

**begin**

**definition**  $tm1 \equiv tm\text{-right-many } \{j + 1, j + 2, j + 4, j + 5\}$

**definition**  $tm2 \equiv tm1 ;; tm\text{-write } (j + 6) \square$

**definition**  $tm3 \equiv tm2 ;; tm\text{-append } (j + 2) (j + 1)$

**definition**  $tm4 \equiv tm3 ;; tm\text{-append } (j + 5) (j + 4)$

**definition**  $tmL1 \equiv tm\text{-simulog } G M j$

**definition**  $tmL2 \equiv tmL1 ;; tm\text{-adjust-headpos } j$

**definition**  $tmL3 \equiv tmL2 ;; tm\text{-adjust-headpos } (j + 3)$

**definition**  $tmL4 \equiv tmL3 ;; tm\text{-write-many } \{j, j + 3\} \triangleright$

**definition**  $tmL \equiv \text{WHILE } [] ; \lambda rs. rs ! (j + 6) < \text{length } M \text{ DO } tmL4 \text{ DONE}$

**definition**  $tm5 \equiv tm4 ;; tmL$

**definition**  $tm6 \equiv tm5 ;; tm\text{-write } (j + 6) \triangleright$

**definition**  $tm7 \equiv tm6 ;; tm\text{-cr } (j + 2)$

**definition**  $tm8 \equiv tm7 ;; tm\text{-cr } (j + 5)$

**lemma**  $tm8\text{-eq-}tm\text{-list-headpos}: tm8 = tm\text{-list-headpos } G M j$

*<proof>*

**context**

**fixes**  $tps0 :: \text{tape list}$

**fixes**  $thalt k :: \text{nat}$  **and**  $xs :: \text{symbol list}$

**assumes**  $M: \text{turing-machine } 2 G M$

**assumes**  $jk: k \geq j + 9 \ j > 0 \ \text{length } tps0 = k$

**assumes**  $thalt:$

$\forall t < thalt. \text{fst } (\text{execute } M \text{ (start-config } 2 \ xs) \ t) < \text{length } M$

$\text{fst } (\text{execute } M \text{ (start-config } 2 \ xs) \ thalt) = \text{length } M$

**assumes**  $xs: \text{symbols-lt } G \ xs$

**assumes**  $tps0:$

$tps0 ! j = [\triangleright]$

$tps0 ! (j + 1) = ([0]_N, 0)$

$tps0 ! (j + 2) = ([\ ]_{NL}, 0)$

$tps0 ! (j + 3) = [\triangleright]$

$tps0 ! (j + 4) = ([0]_N, 0)$

$tps0 ! (j + 5) = ([\ ]_{NL}, 0)$

$tps0 ! (j + 6) = [\triangleright]$

$tps0 ! (j + 7) = ([xs], 0)$

$tps0 ! (j + 8) = ([\ ]_{NL}, 0)$

**begin**

**abbreviation**  $exec :: \text{nat} \Rightarrow \text{config}$  **where**

$exec \ t \equiv \text{execute } M \text{ (start-config } 2 \ xs) \ t$

**lemma**  $max\text{-head-pos-}0: \forall t. \text{exec } t < \# > 0 \leq thalt$

*<proof>*

**lemma**  $max\text{-head-pos-}1: \forall t. \text{exec } t < \# > 1 \leq thalt$

*<proof>*

**definition**  $tps1 \equiv tps0$

$[(j + 1) := ([0]_N, 1),$

$(j + 2) := ([\ ]_{NL}, 1),$

$(j + 4) := ([0]_N, 1),$

$(j + 5) := ([\ ]_{NL}, 1),$

$(j + 6) := [\triangleright]$

**lemma**  $tm1$  [*transforms-intros*]:

**assumes**  $ttt = 1$

**shows**  $\text{transforms } tm1 \ tps0 \ ttt \ tps1$



$j + 8 := \text{exec } (\text{Suc } t) \langle ! \rangle 1]$

**lemma** *lentpsL1*:  $\text{length } (\text{tpsL1 } t) = k$   
*<proof>*

**lemma** *tmL1* [*transforms-intros*]:  
**assumes**  $\text{fst } (\text{exec } t) < \text{length } M$   
**shows** *transforms tmL1* ( $\text{tpsL } t$ ) 1 ( $\text{tpsL1 } t$ )  
*<proof>*

**definition** *tpsL2*  $t \equiv \text{tps0}$   
 $[j := \lceil \text{direction-to-symbol } ((M ! \text{fst } (\text{exec } t)) (\text{config-read } (\text{exec } t)) [\sim] 0) \rceil,$   
 $j + 1 := (\lfloor \text{exec } (\text{Suc } t) \langle \# \rangle 0 \rfloor_N, 1),$   
 $j + 2 := \text{nltape } (\text{map } (\lambda t. \text{exec } t \langle \# \rangle 0) [0..<\text{Suc } (\text{Suc } t)]),$   
 $j + 3 := \lceil \text{direction-to-symbol } ((M ! \text{fst } (\text{exec } t)) (\text{config-read } (\text{exec } t)) [\sim] 1) \rceil,$   
 $j + 4 := (\lfloor \text{exec } t \langle \# \rangle 1 \rfloor_N, 1),$   
 $j + 5 := \text{nltape } (\text{map } (\lambda t. \text{exec } t \langle \# \rangle 1) [0..<\text{Suc } t]),$   
 $j + 6 := \lceil \text{fst } (\text{exec } (\text{Suc } t)) \rceil,$   
 $j + 7 := \text{exec } (\text{Suc } t) \langle ! \rangle 0,$   
 $j + 8 := \text{exec } (\text{Suc } t) \langle ! \rangle 1]$

**lemma** *lentpsL2*:  $\text{length } (\text{tpsL2 } t) = k$   
*<proof>*

**lemma** *tmL2* [*transforms-intros*]:  
**assumes**  $\text{fst } (\text{exec } t) < \text{length } M$   
**and**  $\text{tnt} = 17 + 4 * \text{nlength thalt}$   
**shows** *transforms tmL2* ( $\text{tpsL } t$ )  $\text{tnt}$  ( $\text{tpsL2 } t$ )  
*<proof>*

**definition** *tpsL3*  $t \equiv \text{tps0}$   
 $[j := \lceil \text{direction-to-symbol } ((M ! \text{fst } (\text{exec } t)) (\text{config-read } (\text{exec } t)) [\sim] 0) \rceil,$   
 $j + 1 := (\lfloor \text{exec } (\text{Suc } t) \langle \# \rangle 0 \rfloor_N, 1),$   
 $j + 2 := \text{nltape } (\text{map } (\lambda t. \text{exec } t \langle \# \rangle 0) [0..<\text{Suc } (\text{Suc } t)]),$   
 $j + 3 := \lceil \text{direction-to-symbol } ((M ! \text{fst } (\text{exec } t)) (\text{config-read } (\text{exec } t)) [\sim] 1) \rceil,$   
 $j + 4 := (\lfloor \text{exec } (\text{Suc } t) \langle \# \rangle 1 \rfloor_N, 1),$   
 $j + 5 := \text{nltape } (\text{map } (\lambda t. \text{exec } t \langle \# \rangle 1) [0..<\text{Suc } (\text{Suc } t)]),$   
 $j + 6 := \lceil \text{fst } (\text{exec } (\text{Suc } t)) \rceil,$   
 $j + 7 := \text{exec } (\text{Suc } t) \langle ! \rangle 0,$   
 $j + 8 := \text{exec } (\text{Suc } t) \langle ! \rangle 1]$

**lemma** *lentpsL3*:  $\text{length } (\text{tpsL3 } t) = k$   
*<proof>*

**lemma** *tmL3* [*transforms-intros*]:  
**assumes**  $\text{fst } (\text{exec } t) < \text{length } M$  **and**  $\text{tnt} = 33 + 8 * \text{nlength thalt}$   
**shows** *transforms tmL3* ( $\text{tpsL } t$ )  $\text{tnt}$  ( $\text{tpsL3 } t$ )  
*<proof>*

**definition** *tpsL4*  $t \equiv \text{tps0}$   
 $[j + 1 := (\lfloor \text{exec } (\text{Suc } t) \langle \# \rangle 0 \rfloor_N, 1),$   
 $j + 2 := \text{nltape } (\text{map } (\lambda t. \text{exec } t \langle \# \rangle 0) [0..<\text{Suc } (\text{Suc } t)]),$   
 $j + 4 := (\lfloor \text{exec } (\text{Suc } t) \langle \# \rangle 1 \rfloor_N, 1),$   
 $j + 5 := \text{nltape } (\text{map } (\lambda t. \text{exec } t \langle \# \rangle 1) [0..<\text{Suc } (\text{Suc } t)]),$   
 $j + 6 := \lceil \text{fst } (\text{exec } (\text{Suc } t)) \rceil,$   
 $j + 7 := \text{exec } (\text{Suc } t) \langle ! \rangle 0,$   
 $j + 8 := \text{exec } (\text{Suc } t) \langle ! \rangle 1]$

**lemma** *lentpsL4*:  $\text{length } (\text{tpsL4 } t) = k$   
*<proof>*

**lemma** *tmL4*:  
**assumes**  $\text{fst } (\text{exec } t) < \text{length } M$

**and**  $t_{tt} = 34 + 8 * nlength\ thalt$   
**shows** *transforms*  $tmL4\ (tpsL\ t)\ t_{tt}\ (tpsL4\ t)$   
*<proof>*

**lemma** *tpsL4-Suc*:  $tpsL4\ t = tpsL\ (Suc\ t)$  (**is**  $?l = ?r$ )  
*<proof>*

**lemma** *tmL4'*:  
**assumes**  $fst\ (exec\ t) < length\ M$   
**and**  $t_{tt} = 34 + 8 * nlength\ thalt$   
**shows** *transforms*  $tmL4\ (tpsL\ t)\ t_{tt}\ (tpsL\ (Suc\ t))$   
*<proof>*

**lemma** *tmL*:  
**assumes**  $t_{tt} = thalt * (36 + 8 * nlength\ thalt) + 1$   
**shows** *transforms*  $tmL\ (tpsL\ 0)\ t_{tt}\ (tpsL\ thalt)$   
*<proof>*

**lemma** *tmL'* [*transforms-intros*]:  
**assumes**  $t_{tt} = thalt * (36 + 8 * nlength\ thalt) + 1$   
**shows** *transforms*  $tmL\ tps4\ t_{tt}\ (tpsL\ thalt)$   
*<proof>*

**definition** *tps5*  $\equiv tps0$   
 $[(j + 1) := (\lfloor exec\ thalt\ <\#\>\ 0 \rfloor_N, 1),$   
 $(j + 2) := nltape\ (map\ (\lambda t. exec\ t\ <\#\>\ 0)\ [0..<Suc\ thalt]),$   
 $(j + 4) := (\lfloor exec\ thalt\ <\#\>\ 1 \rfloor_N, 1),$   
 $(j + 5) := nltape\ (map\ (\lambda t. exec\ t\ <\#\>\ 1)\ [0..<Suc\ thalt]),$   
 $(j + 6) := \lceil fst\ (exec\ thalt) \rceil,$   
 $(j + 7) := exec\ thalt\ <!\>\ 0,$   
 $(j + 8) := exec\ thalt\ <!\>\ 1]$

**lemma** *tm5*:  
**assumes**  $t_{tt} = thalt * (36 + 8 * nlength\ thalt) + 15$   
**shows** *transforms*  $tm5\ tps0\ t_{tt}\ (tpsL\ thalt)$   
*<proof>*

**lemma** *tm5'* [*transforms-intros*]:  
**assumes**  $t_{tt} = thalt * (36 + 8 * nlength\ thalt) + 15$   
**shows** *transforms*  $tm5\ tps0\ t_{tt}\ tps5$   
*<proof>*

**definition** *tps6*  $\equiv tps0$   
 $[(j + 1) := (\lfloor exec\ thalt\ <\#\>\ 0 \rfloor_N, 1),$   
 $(j + 2) := nltape\ (map\ (\lambda t. exec\ t\ <\#\>\ 0)\ [0..<Suc\ thalt]),$   
 $(j + 4) := (\lfloor exec\ thalt\ <\#\>\ 1 \rfloor_N, 1),$   
 $(j + 5) := nltape\ (map\ (\lambda t. exec\ t\ <\#\>\ 1)\ [0..<Suc\ thalt]),$   
 $(j + 7) := exec\ thalt\ <!\>\ 0,$   
 $(j + 8) := exec\ thalt\ <!\>\ 1]$

**lemma** *tm6* [*transforms-intros*]:  
**assumes**  $t_{tt} = thalt * (36 + 8 * nlength\ thalt) + 16$   
**shows** *transforms*  $tm6\ tps0\ t_{tt}\ tps6$   
*<proof>*

**definition** *tps7*  $\equiv tps0$   
 $[(j + 1) := (\lfloor exec\ thalt\ <\#\>\ 0 \rfloor_N, 1),$   
 $(j + 2) := (\lfloor map\ (\lambda t. exec\ t\ <\#\>\ 0)\ [0..<Suc\ thalt] \rfloor_{NL}, 1),$   
 $(j + 4) := (\lfloor exec\ thalt\ <\#\>\ 1 \rfloor_N, 1),$   
 $(j + 5) := nltape\ (map\ (\lambda t. exec\ t\ <\#\>\ 1)\ [0..<Suc\ thalt]),$   
 $(j + 7) := exec\ thalt\ <!\>\ 0,$   
 $(j + 8) := exec\ thalt\ <!\>\ 1]$

**lemma** *tm7* [*transforms-intros*]:

**assumes**  $ttt = thalt * (36 + 8 * nlength\ thalt) + 19 + nlength\ (map\ (\lambda t.\ exec\ t\ <\#\>\ 0)\ [0..\<Suc\ thalt])$   
**shows** *transforms tm7 tps0 ttt tps7*  
*<proof>*

**definition** *tps8*  $\equiv$  *tps0*

$[(j + 1) := (\lfloor exec\ thalt\ <\#\>\ 0 \rfloor_N, 1),$   
 $(j + 2) := (\lfloor map\ (\lambda t.\ exec\ t\ <\#\>\ 0) [0..\<Suc\ thalt] \rfloor_{NL}, 1),$   
 $(j + 4) := (\lfloor exec\ thalt\ <\#\>\ 1 \rfloor_N, 1),$   
 $(j + 5) := (\lfloor map\ (\lambda t.\ exec\ t\ <\#\>\ 1) [0..\<Suc\ thalt] \rfloor_{NL}, 1),$   
 $(j + 7) := exec\ thalt\ <!\>\ 0,$   
 $(j + 8) := exec\ thalt\ <!\>\ 1]$

**lemma** *tm8*:

**assumes**  $ttt = thalt * (36 + 8 * nlength\ thalt) + 22 + nlength\ (map\ (\lambda t.\ exec\ t\ <\#\>\ 0)\ [0..\<Suc\ thalt]) +$   
 $nlength\ (map\ (\lambda t.\ (exec\ t)\ <\#\>\ 1)\ [0..\<Suc\ thalt])$   
**shows** *transforms tm8 tps0 ttt tps8*  
*<proof>*

**lemma** *tm8'*:

**assumes**  $ttt = 27 * Suc\ thalt * (9 + 2 * nlength\ thalt)$   
**shows** *transforms tm8 tps0 ttt tps8*  
*<proof>*

**end**

**end**

**lemma** *transforms-tm-list-headposI* [*transforms-intros*]:

**fixes**  $G :: nat$  **and**  $j :: tapeidx$  **and**  $M :: machine$   
**fixes**  $tps\ tps' :: tape\ list$   
**fixes**  $thalt\ k\ ttt :: nat$  **and**  $xs :: symbol\ list$   
**assumes** *turing-machine 2 G M*  
**assumes**  $length\ tps = k$  **and**  $k \geq j + 9$  **and**  $j > 0$   
**assumes**  
 $\forall t < thalt.\ fst\ (execute\ M\ (start-config\ 2\ xs)\ t) < length\ M$   
 $fst\ (execute\ M\ (start-config\ 2\ xs)\ thalt) = length\ M$   
**assumes** *symbols-lt G xs*  
**assumes**  $tps\ !\ j = \lceil \triangleright \rceil$   
 $tps\ !\ (j + 1) = (\lfloor 0 \rfloor_N, 0)$   
 $tps\ !\ (j + 2) = (\lfloor [] \rfloor_{NL}, 0)$   
 $tps\ !\ (j + 3) = \lceil \triangleright \rceil$   
 $tps\ !\ (j + 4) = (\lfloor 0 \rfloor_N, 0)$   
 $tps\ !\ (j + 5) = (\lfloor [] \rfloor_{NL}, 0)$   
 $tps\ !\ (j + 6) = \lceil \triangleright \rceil$   
 $tps\ !\ (j + 7) = (\lfloor xs \rfloor, 0)$   
 $tps\ !\ (j + 8) = (\lfloor [] \rfloor, 0)$   
**assumes**  $ttt = 27 * Suc\ thalt * (9 + 2 * nlength\ thalt)$   
**assumes**  $tps' = tps$   
 $[(j + 1) := (\lfloor (execute\ M\ (start-config\ 2\ xs)\ thalt)\ <\#\>\ 0 \rfloor_N, 1),$   
 $(j + 2) := (\lfloor map\ (\lambda t.\ (execute\ M\ (start-config\ 2\ xs)\ t)\ <\#\>\ 0) [0..\<Suc\ thalt] \rfloor_{NL}, 1),$   
 $(j + 4) := (\lfloor (execute\ M\ (start-config\ 2\ xs)\ thalt)\ <\#\>\ 1 \rfloor_N, 1),$   
 $(j + 5) := (\lfloor map\ (\lambda t.\ (execute\ M\ (start-config\ 2\ xs)\ t)\ <\#\>\ 1) [0..\<Suc\ thalt] \rfloor_{NL}, 1),$   
 $(j + 7) := (execute\ M\ (start-config\ 2\ xs)\ thalt)\ <!\>\ 0,$   
 $(j + 8) := (execute\ M\ (start-config\ 2\ xs)\ thalt)\ <!\>\ 1]$   
**shows** *transforms (tm-list-headpos G M j) tps ttt tps'*  
*<proof>*

## 7.4 A Turing machine for $\Psi$ formulas

CNF formulas from the  $\Psi$  family of formulas feature prominently in  $\Phi$ . In this section we first present a Turing machine for generating arbitrary members of this family and later a specialized one for the  $\Psi$

formulas that we need for  $\Phi$ .

### 7.4.1 The general case

The next Turing machine generates a representation of the CNF formula  $\Psi(vs, k)$ . It expects  $vs$  as a list of numbers on tape  $j$  and the number  $k$  on tape  $j + 1$ . A list of lists of numbers representing  $\Psi(vs, k)$  is output to tape  $j + 2$ .

The TM iterates over the elements of  $vs$ . In each iteration it generates a singleton clause containing the current element of  $vs$  either as positive or negative literal, depending on whether  $k$  is greater than zero or equal to zero. Then it decrements the number  $k$ . Thus the first  $k$  variable indices of  $vs$  will be turned into  $k$  positive literals, the rest into negative ones (provided  $|vs| \geq k$ ).

**definition** *tm-Psi* :: *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-Psi j  $\equiv$ 
  WHILE [] ;  $\lambda rs. rs ! j \neq \square$  DO
    tm-nextract | j (j + 3) ;;
    tm-times2 (j + 3) ;;
    IF  $\lambda rs. rs ! (j + 1) \neq \square$  THEN
      tm-incr (j + 3)
    ELSE
      []
    ENDIF ;;
    tm-to-list (j + 3) ;;
    tm-appendl (j + 2) (j + 3) ;;
    tm-decr (j + 1) ;;
    tm-erase-cr (j + 3)
  DONE ;;
  tm-cr (j + 2) ;;
  tm-erase-cr j

```

**lemma** *tm-Psi-tm*:

```

assumes  $0 < j$  and  $j + 3 < k$  and  $G \geq 6$ 
shows turing-machine k G (tm-Psi j)
<proof>

```

Two lemmas to help with the running time bound of *tm-Psi*:

**lemma** *sum-list-mono-nth*:

```

fixes xs :: 'a list and f g :: 'a  $\Rightarrow$  nat
assumes  $\forall i < \text{length } xs. f (xs ! i) \leq g (xs ! i)$ 
shows sum-list (map f xs)  $\leq$  sum-list (map g xs)
<proof>

```

**lemma** *sum-list-plus-const*:

```

fixes xs :: 'a list and f :: 'a  $\Rightarrow$  nat and c :: nat
shows sum-list (map ( $\lambda x. c + f x$ ) xs) =  $c * \text{length } xs + \text{sum-list (map f xs)}$ 
<proof>

```

**locale** *turing-machine-Psi* =

```

  fixes j :: tapeidx
begin

```

**definition** *tm1*  $\equiv$  *tm-nextract* | j (j + 3)

**definition** *tm2*  $\equiv$  *tm1* ;; *tm-times2* (j + 3)

**definition** *tm23*  $\equiv$  IF  $\lambda rs. rs ! (j + 1) \neq \square$  THEN *tm-incr* (j + 3) ELSE [] ENDIF

**definition** *tm3*  $\equiv$  *tm2* ;; *tm23*

**definition** *tm4*  $\equiv$  *tm3* ;; *tm-to-list* (j + 3)

**definition** *tm5*  $\equiv$  *tm4* ;; *tm-appendl* (j + 2) (j + 3)

**definition** *tm6*  $\equiv$  *tm5* ;; *tm-decr* (j + 1)

**definition** *tm7*  $\equiv$  *tm6* ;; *tm-erase-cr* (j + 3)

**definition** *tmL*  $\equiv$  WHILE [] ;  $\lambda rs. rs ! j \neq \square$  DO *tm7* DONE

**definition** *tm8*  $\equiv$  *tmL* ;; *tm-cr* (j + 2)

**definition**  $tm9 \equiv tm8$  ;; *tm-erase-cr*  $j$

**lemma** *tm9-eq-tm-Psi*:  $tm9 = tm-Psi\ j$   
 ⟨*proof*⟩

**context**

**fixes**  $tps0$  :: *tape list* **and**  $k\ kk$  :: *nat* **and**  $ns$  :: *nat list*  
**assumes**  $jk$ :  $length\ tps0 = k$   $0 < j$   $j + 3 < k$   
**and**  $kk$ :  $kk \leq length\ ns$   
**assumes**  $tps0$ :  
 $tps0\ !\ j = ([ns]_{NL}, 1)$   
 $tps0\ !\ (j + 1) = ([kk]_N, 1)$   
 $tps0\ !\ (j + 2) = ([\ ]_{NLL}, 1)$   
 $tps0\ !\ (j + 3) = ([\ ], 1)$

**begin**

**definition**  $tpsL$  :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL\ t \equiv tps0$   
 $[j := nltape'\ ns\ t,$   
 $j + 1 := ([kk - t]_N, 1),$   
 $j + 2 := nlltape\ (map\ (\lambda t. [2 * ns\ !\ t + (if\ t < kk\ then\ 1\ else\ 0)])\ [0..<t])]$

**lemma** *tpsL-eq-tps0*:  $tpsL\ 0 = tps0$   
 ⟨*proof*⟩

**definition**  $tpsL1$  :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL1\ t \equiv tps0$   
 $[j := nltape'\ ns\ (Suc\ t),$   
 $j + 1 := ([kk - t]_N, 1),$   
 $j + 2 := nlltape\ (map\ (\lambda t. [2 * ns\ !\ t + (if\ t < kk\ then\ 1\ else\ 0)])\ [0..<t]),$   
 $j + 3 := ([ns\ !\ t]_N, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:  
**assumes**  $t < length\ ns$   
**and**  $ttt = 12 + 2 * nlength\ (ns\ !\ t)$   
**shows** *transforms*  $tm1$  ( $tpsL\ t$ )  $ttt$  ( $tpsL1\ t$ )  
 ⟨*proof*⟩

**definition**  $tpsL2$  :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL2\ t \equiv tps0$   
 $[j := nltape'\ ns\ (Suc\ t),$   
 $j + 1 := ([kk - t]_N, 1),$   
 $j + 2 := nlltape\ (map\ (\lambda t. [2 * ns\ !\ t + (if\ t < kk\ then\ 1\ else\ 0)])\ [0..<t]),$   
 $j + 3 := ([2 * ns\ !\ t]_N, 1)]$

**lemma**  $tm2$  [*transforms-intros*]:  
**assumes**  $t < length\ ns$   
**and**  $ttt = 17 + 4 * nlength\ (ns\ !\ t)$   
**shows** *transforms*  $tm2$  ( $tpsL\ t$ )  $ttt$  ( $tpsL2\ t$ )  
 ⟨*proof*⟩

**definition**  $tpsL3$  :: *nat*  $\Rightarrow$  *tape list* **where**

$tpsL3\ t \equiv tps0$   
 $[j := nltape'\ ns\ (Suc\ t),$   
 $j + 1 := ([kk - t]_N, 1),$   
 $j + 2 := nlltape\ (map\ (\lambda t. [2 * ns\ !\ t + (if\ t < kk\ then\ 1\ else\ 0)])\ [0..<t]),$   
 $j + 3 := ([2 * ns\ !\ t + (if\ t < kk\ then\ 1\ else\ 0)]_N, 1)]$

**lemma**  $tm23$  [*transforms-intros*]:  
**assumes**  $t < length\ ns$   
**and**  $ttt = 7 + 2 * nlength\ (2 * ns\ !\ t)$   
**shows** *transforms*  $tm23$  ( $tpsL2\ t$ )  $ttt$  ( $tpsL3\ t$ )  
 ⟨*proof*⟩

**lemma** *tm3*:

**assumes**  $t < \text{length } ns$   
**and**  $t \leq 24 + 4 * \text{nlength } (ns ! t) + 2 * \text{nlength } (2 * ns ! t)$   
**shows** *transforms tm3* (*tpsL*  $t$ ) *t* (*tpsL3*  $t$ )  
(*proof*)

**lemma** *tm3'* [*transforms-intros*]:

**assumes**  $t < \text{length } ns$  **and**  $t \leq 26 + 6 * \text{nlength } (ns ! t)$   
**shows** *transforms tm3* (*tpsL*  $t$ ) *t* (*tpsL3*  $t$ )  
(*proof*)

**definition** *tpsL4* ::  $nat \Rightarrow \text{tape list}$  **where**

$tpsL4\ t \equiv tps0$   
 $[j := nltape' ns (Suc\ t),$   
 $j + 1 := (\lfloor kk - t \rfloor_N, 1),$   
 $j + 2 := nlltape (map (\lambda t. [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)]) [0..<t]),$   
 $j + 3 := (\lfloor [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)] \rfloor_{NL}, 1)]$

**lemma** *tm4*:

**assumes**  $t < \text{length } ns$   
**and**  $t \leq 31 + 6 * \text{nlength } (ns ! t) + 2 * \text{nlength } (2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0))$   
**shows** *transforms tm4* (*tpsL*  $t$ ) *t* (*tpsL4*  $t$ )  
(*proof*)

**lemma** *tm4'* [*transforms-intros*]:

**assumes**  $t < \text{length } ns$  **and**  $t \leq 33 + 8 * \text{nlength } (ns ! t)$   
**shows** *transforms tm4* (*tpsL*  $t$ ) *t* (*tpsL4*  $t$ )  
(*proof*)

**definition** *tpsL5* ::  $nat \Rightarrow \text{tape list}$  **where**

$tpsL5\ t \equiv tps0$   
 $[j := nltape' ns (Suc\ t),$   
 $j + 1 := (\lfloor kk - t \rfloor_N, 1),$   
 $j + 2 := nlltape (map (\lambda t. [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)]) [0..<Suc\ t]),$   
 $j + 3 := (\lfloor [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)] \rfloor_{NL}, 1)]$

**lemma** *tm5* [*transforms-intros*]:

**assumes**  $t < \text{length } ns$   
**and**  $t \leq 39 + 8 * \text{nlength } (ns ! t) + 2 * \text{nlength } [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)]$   
**shows** *transforms tm5* (*tpsL*  $t$ ) *t* (*tpsL5*  $t$ )  
(*proof*)

**definition** *tpsL6* ::  $nat \Rightarrow \text{tape list}$  **where**

$tpsL6\ t \equiv tps0$   
 $[j := nltape' ns (Suc\ t),$   
 $j + 1 := (\lfloor kk - t - 1 \rfloor_N, 1),$   
 $j + 2 := nlltape (map (\lambda t. [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)]) [0..<Suc\ t]),$   
 $j + 3 := (\lfloor [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)] \rfloor_{NL}, 1)]$

**lemma** *tm6*:

**assumes**  $t < \text{length } ns$   
**and**  $t \leq 39 + 8 * \text{nlength } (ns ! t) +$   
 $2 * \text{nlength } [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)] +$   
 $(8 + 2 * \text{nlength } (kk - t))$   
**shows** *transforms tm6* (*tpsL*  $t$ ) *t* (*tpsL6*  $t$ )  
(*proof*)

**lemma** *nllength-elem*:  $\text{nllength } [2 * ns ! t + (if\ t < kk\ then\ 1\ else\ 0)] \leq 2 + \text{nlength } (ns ! t)$   
(*proof*)

**lemma** *tm6'* [*transforms-intros*]:

**assumes**  $t < \text{length } ns$

**and**  $t_{tt} = 43 + 10 * \text{nlength } (ns ! t) + (8 + 2 * \text{nlength } (kk - t))$   
**shows** *transforms tm6* (*tpsL t*) *t\_{tt}* (*tpsL6 t*)  
<proof>

**definition** *tpsL7* :: *nat*  $\Rightarrow$  *tape list* **where**

*tpsL7 t*  $\equiv$  *tps0*  
 $[j := \text{nltape}' ns (\text{Suc } t),$   
 $j + 1 := (\lfloor kk - \text{Suc } t \rfloor_N, 1),$   
 $j + 2 := \text{nlltape} (\text{map } (\lambda t. [2 * ns ! t + (\text{if } t < kk \text{ then } 1 \text{ else } 0)]) [0..<\text{Suc } t]),$   
 $j + 3 := (\lfloor \rfloor, 1)]$

**lemma** *tm7*:

**assumes**  $t < \text{length } ns$   
**and**  $t_{tt} = 51 + (10 * \text{nlength } (ns ! t) + 2 * \text{nlength } (kk - t) +$   
 $(7 + 2 * \text{length } (\text{numlist } [2 * ns ! t + (\text{if } t < kk \text{ then } 1 \text{ else } 0)]))$   
**shows** *transforms tm7* (*tpsL t*) *t\_{tt}* (*tpsL7 t*)  
<proof>

**lemma** *tm7'*:

**assumes**  $t < \text{length } ns$  **and**  $t_{tt} = 62 + 14 * \text{nlength } ns$   
**shows** *transforms tm7* (*tpsL t*) *t\_{tt}* (*tpsL7 t*)  
<proof>

**lemma** *tpsL7-eq-tpsL*: *tpsL7 t* = *tpsL (Suc t)*

<proof>

**lemma** *tm7''* [*transforms-intros*]:

**assumes**  $t < \text{length } ns$  **and**  $t_{tt} = 62 + 14 * \text{nlength } ns$   
**shows** *transforms tm7* (*tpsL t*) *t\_{tt}* (*tpsL (Suc t)*)  
<proof>

**lemma** *tmL* [*transforms-intros*]:

**assumes**  $t_{tt} = \text{length } ns * (62 + 14 * \text{nlength } ns + 2) + 1$   
**shows** *transforms tmL* (*tpsL 0*) *t\_{tt}* (*tpsL (length ns)*)  
<proof>

**definition** *tps8* :: *tape list* **where**

*tps8*  $\equiv$  *tps0*  
 $[j := \text{nltape}' ns (\text{length } ns),$   
 $j + 1 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 2 := \text{nlltape}' (\text{map } (\lambda t. [2 * ns ! t + (\text{if } t < kk \text{ then } 1 \text{ else } 0)]) [0..<\text{length } ns]) 0]$

**lemma** *tm8*:

**assumes**  $t_{tt} = \text{Suc } (\text{length } ns * (64 + 14 * \text{nlength } ns) +$   
 $\text{Suc } (\text{Suc } (\text{Suc } (\text{nllength } (\text{map } (\lambda t. [2 * ns ! t + (\text{if } t < kk \text{ then } 1 \text{ else } 0)]) [0..<\text{length } ns])))$   
**shows** *transforms tm8* (*tpsL 0*) *t\_{tt}* *tps8*  
<proof>

**lemma** *tm8'* [*transforms-intros*]:

**assumes**  $t_{tt} = 4 + 81 * \text{nlength } ns ^ 2$   
**shows** *transforms tm8* *tps0* *t\_{tt}* *tps8*  
<proof>

**definition** *tps9* :: *tape list* **where**

*tps9*  $\equiv$  *tps0*  
 $[j := (\lfloor \rfloor, 1),$   
 $j + 1 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 2 := \text{nlltape}' (\text{map } (\lambda t. [2 * ns ! t + (\text{if } t < kk \text{ then } 1 \text{ else } 0)]) [0..<\text{length } ns]) 0]$

**lemma** *tm9*:

**assumes**  $t_{tt} = 11 + 81 * \text{nlength } ns ^ 2 + 3 * \text{nlength } ns$   
**shows** *transforms tm9* *tps0* *t\_{tt}* *tps9*  
<proof>



*tm-erase-cr* ( $j + 3$ )

**lemma** *tm-Psigamma-tm*:  
**assumes**  $G \geq 6$  **and**  $j + 7 < k$   
**shows** *turing-machine*  $k$   $G$  (*tm-Psigamma*  $j$ )  
*<proof>*

**locale** *turing-machine-Psigamma* =  
  **fixes**  $j :: \text{tapeidx}$   
**begin**

**definition**  $tm1 \equiv \text{tm-mult } j (j + 1) (j + 3)$   
**definition**  $tm2 \equiv tm1 ;; \text{tm-range } (j + 3) (j + 1) (j + 4)$   
**definition**  $tm3 \equiv tm2 ;; \text{tm-copy } (j + 2) (j + 5)$   
**definition**  $tm4 \equiv tm3 ;; \text{tm-Psi } (j + 4)$   
**definition**  $tm5 \equiv tm4 ;; \text{tm-erase-cr } (j + 3)$

**lemma** *tm5-eq-tm-Psigamma*:  $tm5 = \text{tm-Psigamma } j$   
*<proof>*

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $H$   $k$   $idx$   $\kappa :: \text{nat}$   
**assumes**  $jk: \text{length } tps0 = k$   $0 < j$   $j + 7 < k$   
  **and**  $H: H \geq 3$   
  **and**  $\kappa: \kappa \leq H$

**assumes** *tps0*:

$tps0 ! j = ([idx]_N, 1)$   
 $tps0 ! (j + 1) = ([H]_N, 1)$   
 $tps0 ! (j + 2) = ([\kappa]_N, 1)$   
 $tps0 ! (j + 3) = ([[]], 1)$   
 $tps0 ! (j + 4) = ([[]], 1)$   
 $tps0 ! (j + 5) = ([[]], 1)$   
 $tps0 ! (j + 6) = ([[]], 1)$   
 $tps0 ! (j + 7) = ([[]], 1)$

**begin**

**definition**  $tps1 \equiv tps0$   
 $[j + 3 := ([idx * H]_N, 1)]$

**lemma** *tm1 [transforms-intros]*:  
**assumes**  $ttt = 4 + 26 * (\text{nlength } idx + \text{nlength } H) ^ 2$   
**shows** *transforms*  $tm1$   $tps0$   $ttt$   $tps1$   
*<proof>*

**definition**  $tps2 \equiv tps0$   
 $[j + 3 := ([idx * H]_N, 1),$   
 $j + 4 := ([ [idx * H..<idx * H + H] ]_{NL}, 1)]$

**lemma** *tm2 [transforms-intros]*:  
**assumes**  $ttt = 4 + 26 * (\text{nlength } idx + \text{nlength } H) ^ 2 + \text{Suc } H * (43 + 9 * \text{nlength } (idx * H + H))$   
**shows** *transforms*  $tm2$   $tps0$   $ttt$   $tps2$   
*<proof>*

**definition**  $tps3 \equiv tps0$   
 $[j + 3 := ([idx * H]_N, 1),$   
 $j + 4 := ([ [idx * H..<idx * H + H] ]_{NL}, 1),$   
 $j + 5 := ([\kappa]_N, 1)]$

**lemma** *tm3 [transforms-intros]*:  
**assumes**  $ttt = 18 + 26 * (\text{nlength } idx + \text{nlength } H) ^ 2 + \text{Suc } H * (43 + 9 * \text{nlength } (idx * H + H)) + 3 * \text{nlength } \kappa$   
**shows** *transforms*  $tm3$   $tps0$   $ttt$   $tps3$   
*<proof>*

**definition**  $tps4 \equiv tps0$

$[j + 3 := (\lfloor idx * H \rfloor_N, 1),$   
 $j + 4 := (\lfloor \lfloor \rfloor, 1),$   
 $j + 5 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 6 := nlltape' (map (\lambda t. [2 * [idx * H..<idx * H + H] ! t + (if t < \kappa then 1 else 0)])$   
 $[0..<length [idx * H..<idx * H + H]]) 0]$

**lemma**  $tm4$ :

**assumes**  $ttt = 29 + 26 * (nlength\ idx + nlength\ H)^2 + Suc\ H * (43 + 9 * nlength\ (idx * H + H)) +$   
 $3 * nlength\ \kappa + 84 * (nlength\ [idx * H..<idx * H + H])^2$   
**shows**  $transforms\ tm4\ tps0\ ttt\ tps4$   
 $\langle proof \rangle$

**definition**  $tps4' \equiv tps0$

$[j + 3 := (\lfloor idx * H \rfloor_N, 1),$   
 $j + 4 := (\lfloor \lfloor \rfloor, 1),$   
 $j + 5 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 6 := nlltape' (map (\lambda t. [2 * (idx * H + t) + (if t < \kappa then 1 else 0)]) [0..<H]) 0]$

**lemma**  $tps4'-eq-tps4$ :  $tps4' = tps4$

$\langle proof \rangle$

**lemma**  $tm4'$  [transforms-intros]:

**assumes**  $ttt = 29 + 26 * (nlength\ idx + nlength\ H)^2 + Suc\ H * (43 + 9 * nlength\ (idx * H + H)) +$   
 $3 * nlength\ \kappa + 84 * (nlength\ [idx * H..<idx * H + H])^2$   
**shows**  $transforms\ tm4\ tps0\ ttt\ tps4'$   
 $\langle proof \rangle$

**definition**  $tps5 \equiv tps0$

$[j + 3 := (\lfloor \lfloor \rfloor, 1),$   
 $j + 4 := (\lfloor \lfloor \rfloor, 1),$   
 $j + 5 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 6 := nlltape' (map (\lambda t. [2 * (idx * H + t) + (if t < \kappa then 1 else 0)]) [0..<H]) 0]$

**lemma**  $tm5$ :

**assumes**  $ttt = 36 + 26 * (nlength\ idx + nlength\ H)^2 + Suc\ H * (43 + 9 * nlength\ (idx * H + H)) +$   
 $3 * nlength\ \kappa + 84 * (nlength\ [idx * H..<idx * H + H])^2 +$   
 $2 * nlength\ (idx * H)$   
**shows**  $transforms\ tm5\ tps0\ ttt\ tps5$   
 $\langle proof \rangle$

**definition**  $tps5' \equiv tps0$

$[j + 6 := (\lfloor nll-Psi\ (idx * H)\ H\ \kappa \rfloor_{NLL}, 1)]$

**lemma**  $tps5'-eq-tps5$ :  $tps5' = tps5$

$\langle proof \rangle$

**lemma**  $tm5'$ :

**assumes**  $ttt = 1851 * H^4 * (nlength\ (Suc\ idx))^2$   
**shows**  $transforms\ tm5\ tps0\ ttt\ tps5'$   
 $\langle proof \rangle$

**end**

**end**

**lemma**  $transforms-tm-PsigammaI$  [transforms-intros]:

**fixes**  $j :: tape\ idx$   
**fixes**  $tps\ tps' :: tape\ list$  **and**  $ttt\ H\ k\ idx\ \kappa :: nat$   
**assumes**  $length\ tps = k$  **and**  $0 < j$  **and**  $j + 7 < k$   
**and**  $H \geq 3$

```

and  $\kappa \leq H$ 
assumes
   $tps ! j = ([idx]_N, 1)$ 
   $tps ! (j + 1) = ([H]_N, 1)$ 
   $tps ! (j + 2) = ([\kappa]_N, 1)$ 
   $tps ! (j + 3) = ([[]], 1)$ 
   $tps ! (j + 4) = ([[]], 1)$ 
   $tps ! (j + 5) = ([[]], 1)$ 
   $tps ! (j + 6) = ([[]], 1)$ 
   $tps ! (j + 7) = ([[]], 1)$ 
assumes  $ttt = 1851 * H^4 * (nlength (Suc idx))^2$ 
assumes  $tps' = tps$ 
   $[j + 6 := (nll-Psi (idx * H) H \kappa)_{NLL}, 1]$ 
shows  $transforms (tm-Psigamma j) tps ttt tps'$ 
<proof>

```

## 7.5 A Turing machine for $\Upsilon$ formulas

The CNF formula  $\Phi_7$  is made of CNF formulas  $\Upsilon(\gamma_i)$  with  $\gamma_i = [i \cdot H, (i + 1) \cdot H]$ . In this section we build a Turing machine that outputs such CNF formulas.

### 7.5.1 A Turing machine for singleton clauses

The  $\Upsilon$  formulas, just like the  $\Psi$  formulas, are conjunctions of singleton clauses. The next Turing machine outputs singleton clauses. The Turing machine has two parameters: a Boolean *incr* and a tape index *j*. It receives a variable index on tape *j*, a CNF formula as list of lists of numbers on tape *j* + 2 and a number *H* on tape *j* + 3. The TM appends to the formula on tape *j* + 2 a singleton clause with a positive or negative (depending on *incr*) literal derived from the variable index. It also decrements *H* and increments the variable index, which makes it more suitable for use in a loop constructing an  $\Upsilon$  formula. Given our encoding of literals, what the TM actually does is doubling the number on tape *j* + 1 and optionally (if *incr* is true) incrementing it.

**definition**  $tm-times2-appendl :: bool \Rightarrow tapeidx \Rightarrow machine$  **where**

```

 $tm-times2-appendl \ incr \ j \equiv$ 
   $tm-copyn \ j \ (j + 1) \ ;;$ 
   $tm-times2 \ (j + 1) \ ;;$ 
   $(if \ incr \ then \ tm-incr \ (j + 1) \ else \ []) \ ;;$ 
   $tm-to-list \ (j + 1) \ ;;$ 
   $tm-appendl \ (j + 2) \ (j + 1) \ ;;$ 
   $tm-erase-cr \ (j + 1) \ ;;$ 
   $tm-incr \ j \ ;;$ 
   $tm-decr \ (j + 3)$ 

```

**lemma**  $tm-times2-appendl-tm$ :

```

assumes  $0 < j$  and  $j + 3 < k$  and  $G \geq 6$ 
shows  $turing-machine \ k \ G \ (tm-times2-appendl \ incr \ j)$ 
<proof>

```

**locale**  $turing-machine-times2-appendl =$

**fixes**  $j :: tapeidx$

**begin**

**context**

**fixes**  $tps0 :: tape \ list$  **and**  $v \ H \ k :: nat$  **and**  $nss :: nat \ list \ list$  **and**  $incr :: bool$

**assumes**  $jk: length \ tps0 = k \ 0 < j \ j + 3 < k$

**assumes**  $tps0$ :

```

 $tps0 ! j = ([v]_N, 1)$ 
 $tps0 ! (j + 1) = ([[]], 1)$ 
 $tps0 ! (j + 2) = nlltape \ nss$ 
 $tps0 ! (j + 3) = ([H]_N, 1)$ 

```

**begin**

**definition**  $tm1 \equiv tm\text{-copyn } j (j + 1)$   
**definition**  $tm2 \equiv tm1 ;; tm\text{-times2 } (j + 1)$   
**definition**  $tm3 \equiv tm2 ;; (if\ incr\ then\ tm\text{-incr } (j + 1)\ else\ [])$   
**definition**  $tm4 \equiv tm3 ;; tm\text{-to-list } (j + 1)$   
**definition**  $tm5 \equiv tm4 ;; tm\text{-appendl } (j + 2) (j + 1)$   
**definition**  $tm6 \equiv tm5 ;; tm\text{-erase-cr } (j + 1)$   
**definition**  $tm7 \equiv tm6 ;; tm\text{-incr } j$   
**definition**  $tm8 \equiv tm7 ;; tm\text{-decr } (j + 3)$

**lemma**  $tm8\text{-eq-}tm\text{-times2appendl}$ :  $tm8 \equiv tm\text{-times2-appendl incr } j$   
 $\langle proof \rangle$

**definition**  $tps1 \equiv tps0$   
 $[j + 1 := ([v]_N, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:  
**assumes**  $ttt = 14 + 3 * nlength\ v$   
**shows**  $transforms\ tm1\ tps0\ ttt\ tps1$   
 $\langle proof \rangle$

**definition**  $tps2 \equiv tps0$   
 $[j + 1 := ([2 * v]_N, 1)]$

**lemma**  $tm2$  [*transforms-intros*]:  
**assumes**  $ttt = 19 + 5 * nlength\ v$   
**shows**  $transforms\ tm2\ tps0\ ttt\ tps2$   
 $\langle proof \rangle$

**definition**  $tps3 \equiv tps0$   
 $[j + 1 := ([2 * v + (if\ incr\ then\ 1\ else\ 0)]_N, 1)]$

**lemma**  $tm3\text{-True}$ :  
**assumes**  $ttt = 24 + 5 * nlength\ v + 2 * nlength\ (2 * v)$  **and**  $incr$   
**shows**  $transforms\ tm3\ tps0\ ttt\ tps3$   
 $\langle proof \rangle$

**lemma**  $tm3\text{-False}$ :  
**assumes**  $ttt = 19 + 5 * nlength\ v$  **and**  $\neg\ incr$   
**shows**  $transforms\ tm3\ tps0\ ttt\ tps3$   
 $\langle proof \rangle$

**lemma**  $tm3$ :  
**assumes**  $ttt = 24 + 5 * nlength\ v + 2 * nlength\ (2 * v)$   
**shows**  $transforms\ tm3\ tps0\ ttt\ tps3$   
 $\langle proof \rangle$

**lemma**  $tm3'$  [*transforms-intros*]:  
**assumes**  $ttt = 26 + 7 * nlength\ v$   
**shows**  $transforms\ tm3\ tps0\ ttt\ tps3$   
 $\langle proof \rangle$

**definition**  $tps4 \equiv tps0$   
 $[j + 1 := ([2 * v + (if\ incr\ then\ 1\ else\ 0)]_{NL}, 1)]$

**lemma**  $tm4$ :  
**assumes**  $ttt = 31 + 7 * nlength\ v + 2 * nlength\ (2 * v + (if\ incr\ then\ 1\ else\ 0))$   
**shows**  $transforms\ tm4\ tps0\ ttt\ tps4$   
 $\langle proof \rangle$

**lemma**  $tm4'$  [*transforms-intros*]:  
**assumes**  $ttt = 33 + 9 * nlength\ v$   
**shows**  $transforms\ tm4\ tps0\ ttt\ tps4$   
 $\langle proof \rangle$

**definition**  $tps5 \equiv tps0$

$[j + 1 := (\lfloor 2 * v + (\text{if incr then } 1 \text{ else } 0) \rfloor)_{NL}, 1),$   
 $j + 2 := \text{nlltape } (nss @ [\lfloor 2 * v + (\text{if incr then } 1 \text{ else } 0) \rfloor])$

**lemma**  $tm5$  [transforms-intros]:

**assumes**  $ttt = 39 + 9 * \text{nlength } v + 2 * \text{nlength } [2 * v + (\text{if incr then } 1 \text{ else } 0)]$

**shows**  $\text{transforms } tm5 \ tps0 \ ttt \ tps5$

$\langle \text{proof} \rangle$

**definition**  $tps6 \equiv tps0$

$[j + 2 := \text{nlltape } (nss @ [\lfloor 2 * v + (\text{if incr then } 1 \text{ else } 0) \rfloor])$

**lemma**  $tm6$ :

**assumes**  $ttt = 46 + 9 * \text{nlength } v + 4 * \text{nlength } [2 * v + (\text{if incr then } 1 \text{ else } 0)]$

**shows**  $\text{transforms } tm6 \ tps0 \ ttt \ tps6$

$\langle \text{proof} \rangle$

**lemma**  $tm6'$  [transforms-intros]:

**assumes**  $ttt = 54 + 13 * \text{nlength } v$

**shows**  $\text{transforms } tm6 \ tps0 \ ttt \ tps6$

$\langle \text{proof} \rangle$

**definition**  $tps7 \equiv tps0$

$[j := (\lfloor \text{Suc } v \rfloor)_N, 1),$

$j + 2 := \text{nlltape } (nss @ [\lfloor 2 * v + (\text{if incr then } 1 \text{ else } 0) \rfloor])$

**lemma**  $tm7$  [transforms-intros]:

**assumes**  $ttt = 59 + 15 * \text{nlength } v$

**shows**  $\text{transforms } tm7 \ tps0 \ ttt \ tps7$

$\langle \text{proof} \rangle$

**definition**  $tps8 \equiv tps0$

$[j := (\lfloor \text{Suc } v \rfloor)_N, 1),$

$j + 2 := \text{nlltape } (nss @ [\lfloor 2 * v + (\text{if incr then } 1 \text{ else } 0) \rfloor]),$

$j + 3 := (\lfloor H - 1 \rfloor)_N, 1]$

**lemma**  $tm8$ :

**assumes**  $ttt = 67 + 15 * \text{nlength } v + 2 * \text{nlength } H$

**shows**  $\text{transforms } tm8 \ tps0 \ ttt \ tps8$

$\langle \text{proof} \rangle$

**end**

**end**

**lemma**  $\text{transforms-tm-times2-appendII}$  [transforms-intros]:

**fixes**  $j :: \text{tapeid}$  **and**  $\text{incr} :: \text{bool}$

**fixes**  $tps \ tps' :: \text{tape list}$  **and**  $ttt \ v \ H \ k :: \text{nat}$  **and**  $nss :: \text{nat list list}$

**assumes**  $\text{length } tps = k$  **and**  $0 < j$  **and**  $j + 3 < k$

**assumes**

$tps ! j = (\lfloor v \rfloor)_N, 1)$

$tps ! (j + 1) = (\lfloor [] \rfloor), 1)$

$tps ! (j + 2) = \text{nlltape } nss$

$tps ! (j + 3) = (\lfloor H \rfloor)_N, 1)$

**assumes**  $ttt = 67 + 15 * \text{nlength } v + 2 * \text{nlength } H$

**assumes**  $tps' = tps$

$[j := (\lfloor \text{Suc } v \rfloor)_N, 1),$

$j + 2 := \text{nlltape } (nss @ [\lfloor 2 * v + (\text{if incr then } 1 \text{ else } 0) \rfloor]),$

$j + 3 := (\lfloor H - 1 \rfloor)_N, 1)]$

**shows**  $\text{transforms } (\text{tm-times2-appendI } \text{incr } j) \ tps \ ttt \ tps'$

$\langle \text{proof} \rangle$

## 7.5.2 A Turing machine for $\Upsilon(\gamma_i)$ formulas

We will not need the general  $\Upsilon$  formulas, but only  $\Upsilon(\gamma_i)$  for  $\gamma_i = [i \cdot H, (i + 1) \cdot H]$ . Represented as list of lists of numbers they look like this (for  $H \geq 3$ ):

**definition** *nll-Upsilon* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat list list* **where**

*nll-Upsilon* *idx len*  $\equiv$   $[[2 * (idx * len) + 1], [2 * (idx * len + 1) + 1]] @ map (\lambda i. [2 * (idx * len + i)]) [3..<len]$

**lemma** *nll-Upsilon*:

**assumes**  $len \geq 3$

**shows** *nll-Upsilon* *idx len* = *formula-n* ( $\Upsilon$  [*idx\*len..<idx\*len+len*])

(**is** ?*lhs* = ?*rhs*)

*<proof>*

**lemma** *nlllength-nll-Upsilon-le*:

**assumes**  $len \geq 3$

**shows** *nlllength* (*nll-Upsilon* *idx len*)  $\leq len * (4 + nlength\ idx + nlength\ len)$

*<proof>*

The next Turing machine outputs CNF formulas of the shape  $\Upsilon(\gamma_i)$ , where  $\gamma_i = [i \cdot H, (i + 1) \cdot H]$ . It expects a number  $i$  on tape  $j$  and a number  $H$  on tape  $j + 1$ . It writes a representation of the formula to tape  $j + 4$ .

**definition** *tm-Upsilongamma* :: *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-Upsilongamma* *j*  $\equiv$

*tm-copyn* ( $j + 1$ ) ( $j + 5$ ) ;;

*tm-mult* *j* ( $j + 1$ ) ( $j + 2$ ) ;;

*tm-times2-appendl* *True* ( $j + 2$ ) ;;

*tm-times2-appendl* *True* ( $j + 2$ ) ;;

*tm-decr* ( $j + 5$ ) ;;

*tm-incr* ( $j + 2$ ) ;;

*WHILE* [] ;  $\lambda rs. rs ! (j + 5) \neq \square$  *DO*

*tm-times2-appendl* *False* ( $j + 2$ )

*DONE* ;;

*tm-erase-cr* ( $j + 2$ ) ;;

*tm-cr* ( $j + 4$ )

**lemma** *tm-Upsilongamma-tm*:

**assumes**  $0 < j$  **and**  $j + 5 < k$  **and**  $G \geq 6$

**shows** *turing-machine* *k G* (*tm-Upsilongamma* *j*)

*<proof>*

**locale** *turing-machine-Upsilongamma* =

**fixes** *j* :: *tapeidx*

**begin**

**definition** *tm1*  $\equiv$  *tm-copyn* ( $j + 1$ ) ( $j + 5$ )

**definition** *tm2*  $\equiv$  *tm1* ;; *tm-mult* *j* ( $j + 1$ ) ( $j + 2$ )

**definition** *tm3*  $\equiv$  *tm2* ;; *tm-times2-appendl* *True* ( $j + 2$ )

**definition** *tm4*  $\equiv$  *tm3* ;; *tm-times2-appendl* *True* ( $j + 2$ )

**definition** *tm5*  $\equiv$  *tm4* ;; *tm-decr* ( $j + 5$ )

**definition** *tm6*  $\equiv$  *tm5* ;; *tm-incr* ( $j + 2$ )

**definition** *tmB*  $\equiv$  *tm-times2-appendl* *False* ( $j + 2$ )

**definition** *tmL*  $\equiv$  *WHILE* [] ;  $\lambda rs. rs ! (j + 5) \neq \square$  *DO* *tmB* *DONE*

**definition** *tm7*  $\equiv$  *tm6* ;; *tmL*

**definition** *tm8*  $\equiv$  *tm7* ;; *tm-erase-cr* ( $j + 2$ )

**definition** *tm9*  $\equiv$  *tm8* ;; *tm-cr* ( $j + 4$ )

**lemma** *tm9-eq-tm-Upsilongamma*: *tm9* = *tm-Upsilongamma* *j*

*<proof>*

**context**

**fixes** *tps0* :: *tape list* **and** *idx H k* :: *nat*

**assumes** *jk*: *length* *tps0* = *k*  $0 < j$   $j + 5 < k$

**and**  $H: H \geq 3$   
**assumes**  $tps0$ :  
 $tps0 ! j = ([idx]_N, 1)$   
 $tps0 ! (j + 1) = ([H]_N, 1)$   
 $tps0 ! (j + 2) = ([\ ], 1)$   
 $tps0 ! (j + 3) = ([\ ], 1)$   
 $tps0 ! (j + 4) = ([\ ], 1)$   
 $tps0 ! (j + 5) = ([\ ], 1)$   
**begin**

**definition**  $tps1 \equiv tps0$   
 $[j + 5 := ([H]_N, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:  
**assumes**  $ttt = 14 + 3 * nlength H$   
**shows** *transforms*  $tm1 tps0 ttt tps1$   
*<proof>*

**definition**  $tps2 \equiv tps0$   
 $[j + 5 := ([H]_N, 1),$   
 $j + 2 := ([idx * H]_N, 1)]$

**lemma**  $tm2$  [*transforms-intros*]:  
**assumes**  $ttt = 18 + 3 * nlength H + 26 * (nlength idx + nlength H)^2$   
**shows** *transforms*  $tm2 tps0 ttt tps2$   
*<proof>*

**definition**  $tps3 \equiv tps0$   
 $[j + 5 := ([H - 1]_N, 1),$   
 $j + 4 := nlltape ([[2 * (idx * H) + 1]]),$   
 $j + 2 := ([idx * H + 1]_N, 1)]$

**lemma**  $tm3$  [*transforms-intros*]:  
**assumes**  $ttt = 85 + 5 * nlength H + 26 * (nlength idx + nlength H)^2 + 15 * nlength (idx * H)$   
**shows** *transforms*  $tm3 tps0 ttt tps3$   
*<proof>*

**definition**  $tps4 \equiv tps0$   
 $[j + 5 := ([H - 2]_N, 1),$   
 $j + 4 := nlltape ([[2 * (idx * H) + 1], [2 * (idx * H + 1) + 1]]),$   
 $j + 2 := ([idx * H + 2]_N, 1)]$

**lemma**  $tm4$  [*transforms-intros*]:  
**assumes**  $ttt = 152 + 5 * nlength H + 26 * (nlength idx + nlength H)^2 + 15 * nlength (idx * H) +$   
 $15 * nlength (Suc (idx * H)) + 2 * nlength (H - 1)$   
**shows** *transforms*  $tm4 tps0 ttt tps4$   
*<proof>*

**definition**  $tps5 \equiv tps0$   
 $[j + 5 := ([H - 3]_N, 1),$   
 $j + 4 := nlltape ([[2 * (idx * H) + 1], [2 * (idx * H + 1) + 1]]),$   
 $j + 2 := ([idx * H + 2]_N, 1)]$

**lemma**  $tm5$  [*transforms-intros*]:  
**assumes**  $ttt = 160 + 5 * nlength H + 26 * (nlength idx + nlength H)^2 + 15 * nlength (idx * H) +$   
 $15 * nlength (Suc (idx * H)) + 2 * nlength (H - 1) + 2 * nlength (H - 2)$   
**shows** *transforms*  $tm5 tps0 ttt tps5$   
*<proof>*

**definition**  $tps6 \equiv tps0$   
 $[j + 5 := ([H - 3]_N, 1),$   
 $j + 4 := nlltape ([[2 * (idx * H) + 1], [2 * (idx * H + 1) + 1]]),$   
 $j + 2 := ([idx * H + 3]_N, 1)]$

**lemma** *tm6*:

**assumes**  $ttt = 165 + 5 * nlength\ H + 26 * (nlength\ idx + nlength\ H)^2 + 15 * nlength\ (idx * H) + 15 * nlength\ (Suc\ (idx * H)) + 2 * nlength\ (H - 1) + 2 * nlength\ (H - 2) + 2 * nlength\ (Suc\ (Suc\ (idx * H)))$   
**shows** *transforms tm6 tps0 ttt tps6*  
*<proof>*

**lemma** *tm6'* [*transforms-intros*]:

**assumes**  $ttt = 165 + 41 * nlength\ (Suc\ idx * H) + 26 * (nlength\ idx + nlength\ H)^2$   
**shows** *transforms tm6 tps0 ttt tps6*  
*<proof>*

**definition** *tpsL t*  $\equiv$  *tps0*

$[j + 5 := (\lfloor H - 3 - t \rfloor_N, 1),$   
 $j + 4 := nlltape\ ([2 * (idx * H) + 1], [2 * (idx * H + 1) + 1]) @ map\ (\lambda i. [2 * (idx * H + i)]) [3..<3 + t]),$   
 $j + 2 := (\lfloor idx * H + 3 + t \rfloor_N, 1)]$

**lemma** *tpsL-eq-tps6*: *tpsL 0 = tps6*

*<proof>*

**lemma** *map-Suc-append*:  $a \leq b \implies map\ f\ [a..<Suc\ b] = map\ f\ [a..<b] @ [f\ b]$

*<proof>*

**lemma** *tmB*:

**assumes**  $ttt = 67 + 15 * nlength\ (idx * H + 3 + t) + 2 * nlength\ (H - 3 - t)$   
**shows** *transforms tmB (tpsL t) ttt (tpsL (Suc t))*  
*<proof>*

**lemma** *tmB'* [*transforms-intros*]:

**assumes**  $ttt = 67 + 15 * nlength\ (Suc\ idx * H) + 2 * nlength\ H$   
**and**  $t < H - 3$   
**shows** *transforms tmB (tpsL t) ttt (tpsL (Suc t))*  
*<proof>*

**lemma** *tmL*:

**assumes**  $ttt = H * (70 + 17 * nlength\ (Suc\ idx * H))$   
**shows** *transforms tmL (tpsL 0) ttt (tpsL (H - 3))*  
*<proof>*

**definition** *tps7*  $\equiv$  *tps0*

$[j + 5 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 4 := nlltape\ ([2 * (idx * H) + 1], [2 * (idx * H + 1) + 1]) @ map\ (\lambda i. [2 * (idx * H + i)]) [3..<H]),$   
 $j + 2 := (\lfloor Suc\ idx * H \rfloor_N, 1)]$

**lemma** *tpsL-eq-tps7*: *tpsL (H - 3) = tps7*

*<proof>*

**lemma** *tmL'* [*transforms-intros*]:

**assumes**  $ttt = H * (70 + 17 * nlength\ (Suc\ idx * H))$   
**shows** *transforms tmL tps6 ttt tps7*  
*<proof>*

**lemma** *tm7* [*transforms-intros*]:

**assumes**  $ttt = 165 + 41 * nlength\ (H + idx * H) + 26 * (nlength\ idx + nlength\ H)^2 + H * (70 + 17 * nlength\ (H + idx * H))$   
**shows** *transforms tm7 tps0 ttt tps7*  
*<proof>*

**definition** *tps8*  $\equiv$  *tps0*

$[j + 5 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 4 := nlltape\ ([2 * (idx * H) + 1], [2 * (idx * H + 1) + 1]) @ map\ (\lambda i. [2 * (idx * H + i)]) [3..<H]),$

$j + 2 := ([\ ], 1)$

**lemma** *tm8*:

**assumes**  $ttt = 172 + 43 * nlength (H + idx * H) + 26 * (nlength idx + nlength H)^2 + H * (70 + 17 * nlength (H + idx * H))$   
**shows** *transforms tm8 tps0 ttt tps8*  
 $\langle proof \rangle$

**definition**  $tps8' \equiv tps0[j + 4 := nlltape (nll-Upsilon idx H)]$

**lemma** *tps8'-eq-tps8*:  $tps8' = tps8$   
 $\langle proof \rangle$

**lemma** *tm8' [transforms-intros]*:

**assumes**  $ttt = 199 * H * (nlength idx + nlength H)^2$   
**shows** *transforms tm8 tps0 ttt tps8'*  
 $\langle proof \rangle$

**definition**  $tps9 \equiv tps0$

$[j + 4 := ([nll-Upsilon idx H]_{NLL}, 1)]$

**lemma** *tm9*:

**assumes**  $ttt = 199 * H * (nlength idx + nlength H)^2 + Suc (Suc (Suc (nlllength (nll-Upsilon idx H))))$   
**shows** *transforms tm9 tps0 ttt tps9*  
 $\langle proof \rangle$

**lemma** *tm9' [transforms-intros]*:

**assumes**  $ttt = 205 * H * (nlength idx + nlength H)^2$   
**shows** *transforms tm9 tps0 ttt tps9*  
 $\langle proof \rangle$

**end**

**end**

**lemma** *transforms-tm-UpsilongammaI [transforms-intros]*:

**fixes**  $j :: tapeidx$   
**fixes**  $tps tps' :: tape list$  **and**  $ttt idx H k :: nat$   
**assumes**  $length tps = k$  **and**  $0 < j$  **and**  $j + 5 < k$   
**and**  $H \geq 3$   
**assumes**  
 $tps ! j = ([idx]_N, 1)$   
 $tps ! (j + 1) = ([H]_N, 1)$   
 $tps ! (j + 2) = ([\ ], 1)$   
 $tps ! (j + 3) = ([\ ], 1)$   
 $tps ! (j + 4) = ([\ ], 1)$   
 $tps ! (j + 5) = ([\ ], 1)$   
**assumes**  $ttt = 205 * H * (nlength idx + nlength H)^2$   
**assumes**  $tps' = tps[j + 4 := ([nll-Upsilon idx H]_{NLL}, 1)]$   
**shows** *transforms (tm-Upsilongamma j) tps ttt tps'*  
 $\langle proof \rangle$

**end**

## 7.6 Turing machines for the parts of $\Phi$

**theory** *Sat-TM-CNF*

**imports** *Aux-TM-Reducing*

**begin**

In this section we build Turing machines for all parts  $\Phi_0, \dots, \Phi_9$  of the CNF formula  $\Phi$ . Some of them ( $\Phi_0, \Phi_1, \Phi_2$ , and  $\Phi_8$ ) are just fixed-length sequences of  $\Psi$  formulas constructible by fixed-length sequences of *tm-Psigamma* machines. Others ( $\Phi_3, \Phi_4, \Phi_5, \Phi_6$ ) are variable-length and require looping

over a *tm-Psigamma* machine. The TM for  $\Phi_7$  is a loop over *tm-Upsilongamma*. Lastly, the TM for  $\Phi_9$  is a loop over a TM that generates the formulas  $\chi_t$ .

Ideally we would want to prove the semantics of the TMs inside the locale *reduction-sat*, in which  $\Phi$  was defined. However, we use locales to prove the semantics of TMs, and locales cannot be nested. For this reason we have to define the TMs on the theory level and prove their semantics there, too, just as we have done with all TMs until now. In the next chapter the semantics lemmas will be transferred to the locale *reduction-sat*.

Unlike most TMs so far, the TMs in this section are not meant to be reusable but serve a special purpose, namely to be combined into one large TM computing  $\Phi$ . For this reason the TMs are somewhat peculiar. For example, they write their output to the fixed tape 1 rather than having a parameter for the output tape. They also often expect the tapes to be initialized in a very special way. Moreover, the TMs often leave the work tapes in a “dirty” state with remnants of intermediate calculations. The combined TM for all of  $\Phi$  will simply allocate a new batch of work tapes for every individual TM.

### 7.6.1 A Turing machine for $\Phi_0$

The next Turing machine expects a number  $i$  on tape  $j$  and a number  $H$  on tape  $j+1$  and outputs to tape 1 the formula  $\Psi([i \dots H, (i+1) \dots H], 1) \wedge \Psi([(i+1) \dots H, (i+2) \dots H], 1) \wedge \Psi([(i+2) \dots H, (i+3) \dots H], 0)$ , which is just  $\Phi_0$  for suitable values of  $i$  and  $H$ .

**definition** *tm-PHI0* :: *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-PHI0 j  $\equiv$ 
  tm-setn (j + 2) 1 ;;
  tm-Psigamma j ;;
  tm-extendl-erase 1 (j + 6) ;;
  tm-incr j ;;
  tm-Psigamma j ;;
  tm-extendl-erase 1 (j + 6) ;;
  tm-incr j ;;
  tm-setn (j + 2) 0 ;;
  tm-Psigamma j ;;
  tm-extendl 1 (j + 6)

```

**lemma** *tm-PHI0-tm*:

```

assumes 0 < j and j + 8 < k and G  $\geq$  6
shows turing-machine k G (tm-PHI0 j)
<proof>

```

**locale** *turing-machine-PHI0* =

```

fixes j :: tapeidx

```

**begin**

```

definition tm1  $\equiv$  tm-setn (j + 2) 1
definition tm2  $\equiv$  tm1 ;; tm-Psigamma j
definition tm3  $\equiv$  tm2 ;; tm-extendl-erase 1 (j + 6)
definition tm5  $\equiv$  tm3 ;; tm-incr j
definition tm6  $\equiv$  tm5 ;; tm-Psigamma j
definition tm7  $\equiv$  tm6 ;; tm-extendl-erase 1 (j + 6)
definition tm9  $\equiv$  tm7 ;; tm-incr j
definition tm10  $\equiv$  tm9 ;; tm-setn (j + 2) 0
definition tm11  $\equiv$  tm10 ;; tm-Psigamma j
definition tm12  $\equiv$  tm11 ;; tm-extendl 1 (j + 6)

```

**lemma** *tm12-eq-tm-PHI0*: *tm12* = *tm-PHI0* j

```

<proof>

```

**context**

```

fixes tps0 :: tape list and k idx H :: nat
assumes jk: length tps0 = k 1 < j j + 8 < k
and H: H  $\geq$  3
assumes tps0:
  tps0 ! 1 = ([[]], 1)

```

```

    tps0 ! j = ([idx]N, 1)
    tps0 ! (j + 1) = ([H]N, 1)
    tps0 ! (j + 2) = ([[]], 1)
    tps0 ! (j + 3) = ([[]], 1)
    tps0 ! (j + 4) = ([[]], 1)
    tps0 ! (j + 5) = ([[]], 1)
    tps0 ! (j + 6) = ([[]], 1)
    tps0 ! (j + 7) = ([[]], 1)
    tps0 ! (j + 8) = ([[]], 1)
begin

definition tps1 ≡ tps0
[j + 2 := ([1]N, 1)]

lemma tm1 [transforms-intros]:
assumes ttt = 12
shows transforms tm1 tps0 ttt tps1
⟨proof⟩

definition tps2 ≡ tps0
[j + 2 := ([1]N, 1),
j + 6 := ([nll-Psi (idx * H) H 1]NLL, 1)]

lemma tm2 [transforms-intros]:
assumes ttt = 12 + 1851 * H ^ 4 * (nlength (Suc idx))2
shows transforms tm2 tps0 ttt tps2
⟨proof⟩

definition tps3 ≡ tps0
[j + 2 := ([1]N, 1),
1 := nlltape (nll-Psi (idx * H) H (Suc 0)),
j + 6 := ([[]], 1)]

lemma tm3 [transforms-intros]:
assumes ttt = 23 + 1851 * H ^ 4 * (nlength (Suc idx))2 +
4 * nlllength (nll-Psi (idx * H) H (Suc 0))
shows transforms tm3 tps0 ttt tps3
⟨proof⟩

definition tps5 ≡ tps0
[j + 2 := ([1]N, 1),
1 := nlltape (nll-Psi (idx * H) H (Suc 0)),
j + 6 := ([[]], 1),
j := ([Suc idx]N, 1)]

lemma tm5 [transforms-intros]:
assumes ttt = 28 + 1851 * H ^ 4 * (nlength (Suc idx))2 +
4 * nlllength (nll-Psi (idx * H) H 1) +
2 * nlength idx
shows transforms tm5 tps0 ttt tps5
⟨proof⟩

definition tps6 ≡ tps0
[j := ([Suc idx]N, 1),
j + 2 := ([1]N, 1),
j + 6 := ([nll-Psi (Suc idx * H) H (Suc 0)]NLL, 1),
1 := nlltape (nll-Psi (idx * H) H 1)]

lemma tm6 [transforms-intros]:
assumes ttt = 28 + 1851 * H ^ 4 * (nlength (Suc idx))2 +
4 * nlllength (nll-Psi (idx * H) H 1) + 2 * nlength idx +
1851 * H ^ 4 * (nlength (Suc (Suc idx)))2
shows transforms tm6 tps0 ttt tps6

```

*<proof>*

**definition**  $tps7 \equiv tps0$

$[j := (\lfloor Suc\ idx \rfloor_N, 1),$

$j + 2 := (\lfloor 1 \rfloor_N, 1),$

$j + 6 := (\lfloor [] \rfloor, 1),$

$1 := nlltape (nll-Psi (idx * H) H 1 @ nll-Psi (H + idx * H) H 1)]$

**lemma**  $tm7$  [*transforms-intros*]:

**assumes**  $ttt = 39 + 1851 * H^4 * (nlength (Suc\ idx))^2 +$

$4 * nlllength (nll-Psi (idx * H) H 1) +$

$2 * nlength\ idx + 1851 * H^4 * (nlength (Suc (Suc\ idx)))^2 +$

$4 * nlllength (nll-Psi (H + idx * H) H 1)$

**shows** *transforms*  $tm7\ tps0\ ttt\ tps7$

*<proof>*

**definition**  $tps9 \equiv tps0$

$[j := (\lfloor Suc (Suc\ idx) \rfloor_N, 1),$

$j + 2 := (\lfloor 1 \rfloor_N, 1),$

$j + 6 := (\lfloor [] \rfloor, 1),$

$1 := nlltape (nll-Psi (idx * H) H 1 @ nll-Psi (H + idx * H) H 1)]$

**lemma**  $tm9$  [*transforms-intros*]:

**assumes**  $ttt = 44 + 1851 * H^4 * (nlength (Suc\ idx))^2 +$

$4 * nlllength (nll-Psi (idx * H) H 1) + 2 * nlength\ idx +$

$1851 * H^4 * (nlength (Suc (Suc\ idx)))^2 + 4 * nlllength (nll-Psi (H + idx * H) H 1) +$

$2 * nlength (Suc\ idx)$

**shows** *transforms*  $tm9\ tps0\ ttt\ tps9$

*<proof>*

**definition**  $tps10 \equiv tps0$

$[j := (\lfloor Suc (Suc\ idx) \rfloor_N, 1),$

$j + 2 := (\lfloor 0 \rfloor_N, 1),$

$j + 6 := (\lfloor [] \rfloor, 1),$

$1 := nlltape (nll-Psi (idx * H) H 1 @ nll-Psi (H + idx * H) H 1)]$

**lemma**  $tm10$  [*transforms-intros*]:

**assumes**  $ttt = 56 + 1851 * H^4 * (nlength (Suc\ idx))^2 +$

$4 * nlllength (nll-Psi (idx * H) H 1) + 2 * nlength\ idx +$

$1851 * H^4 * (nlength (Suc (Suc\ idx)))^2 + 4 * nlllength (nll-Psi (H + idx * H) H 1) +$

$2 * nlength (Suc\ idx)$

**shows** *transforms*  $tm10\ tps0\ ttt\ tps10$

*<proof>*

**definition**  $tps11 \equiv tps0$

$[j := (\lfloor Suc (Suc\ idx) \rfloor_N, 1),$

$j + 2 := (\lfloor 0 \rfloor_N, 1),$

$j + 6 := (\lfloor nll-Psi (Suc (Suc\ idx) * H) H 0 \rfloor_{NLL}, 1),$

$1 := nlltape (nll-Psi (idx * H) H 1 @ nll-Psi (H + idx * H) H 1)]$

**lemma**  $tm11$  [*transforms-intros*]:

**assumes**  $ttt = 56 + 1851 * H^4 * (nlength (Suc\ idx))^2 +$

$4 * nlllength (nll-Psi (idx * H) H 1) + 2 * nlength\ idx +$

$1851 * H^4 * (nlength (Suc (Suc\ idx)))^2 + 4 * nlllength (nll-Psi (H + idx * H) H 1) +$

$2 * nlength (Suc\ idx) + 1851 * H^4 * (nlength (Suc (Suc (Suc\ idx))))^2$

**shows** *transforms*  $tm11\ tps0\ ttt\ tps11$

*<proof>*

**definition**  $tps12 \equiv tps0$

$[j := (\lfloor Suc (Suc\ idx) \rfloor_N, 1),$

$j + 2 := (\lfloor 0 \rfloor_N, 1),$

$j + 6 := (\lfloor nll-Psi (Suc (Suc\ idx) * H) H 0 \rfloor_{NLL}, 1),$

$1 := nlltape (nll-Psi (idx * H) H 1 @ nll-Psi (H + idx * H) H 1 @ nll-Psi (Suc (Suc\ idx) * H) H 0)]$

**lemma** *tm12*:

**assumes**  $ttt = 60 + 1851 * H^4 * (nlength (Suc\ idx))^2 +$   
 $4 * nlllength (nll-Psi (idx * H) H\ 1) + 2 * nlength\ idx +$   
 $1851 * H^4 * (nlength (Suc (Suc\ idx)))^2 + 4 * nlllength (nll-Psi (H + idx * H) H\ 1) +$   
 $2 * nlength (Suc\ idx) + 1851 * H^4 * (nlength (Suc (Suc (Suc\ idx))))^2 +$   
 $2 * nlllength (nll-Psi (H + (H + idx * H)) H\ 0)$   
**shows** *transforms tm12 tps0 ttt tps12*  
*<proof>*

**lemma** *tm12'*:

**assumes**  $ttt = 5627 * H^4 * (3 + nlength (3 * H + idx * H))^2$   
**shows** *transforms tm12 tps0 ttt tps12*  
*<proof>*

**end**

**end**

**lemma** *transforms-tm-PHI0I*:

**fixes**  $j :: tapeidx$   
**fixes**  $tps\ tps' :: tape\ list$  **and**  $ttt\ k\ idx\ H :: nat$   
**assumes**  $length\ tps = k$  **and**  $1 < j$  **and**  $j + 8 < k$  **and**  $H \geq 3$   
**assumes**  
 $tps\ !\ 1 = ([\ ], 1)$   
 $tps\ !\ j = ([idx]_N, 1)$   
 $tps\ !\ (j + 1) = ([H]_N, 1)$   
 $tps\ !\ (j + 2) = ([\ ], 1)$   
 $tps\ !\ (j + 3) = ([\ ], 1)$   
 $tps\ !\ (j + 4) = ([\ ], 1)$   
 $tps\ !\ (j + 5) = ([\ ], 1)$   
 $tps\ !\ (j + 6) = ([\ ], 1)$   
 $tps\ !\ (j + 7) = ([\ ], 1)$   
 $tps\ !\ (j + 8) = ([\ ], 1)$   
**assumes**  $tps' = tps$   
 $[j := ([Suc (Suc\ idx)]_N, 1),$   
 $j + 2 := ([0]_N, 1),$   
 $j + 6 := ([nll-Psi (Suc (Suc\ idx) * H) H\ 0]_{NLL}, 1),$   
 $1 := nlltape (nll-Psi (idx * H) H\ 1 @ nll-Psi (H + idx * H) H\ 1 @ nll-Psi (Suc (Suc\ idx) * H) H\ 0)]$   
**assumes**  $ttt = 5627 * H^4 * (3 + nlength (3 * H + idx * H))^2$   
**shows** *transforms (tm-PHI0 j) tps ttt tps'*  
*<proof>*

## 7.6.2 A Turing machine for $\Phi_1$

The next TM expects a number  $H$  on tape  $j + 1$  and appends to the formula on tape 1 the formula  $\Psi([0, H], 1)$ .

**definition** *tm-PHI1*  $:: tapeidx \Rightarrow machine$  **where**

*tm-PHI1*  $j \equiv$   
 $tm-setn (j + 2)\ 1\ ;;$   
 $tm-Psigamma\ j\ ;;$   
 $tm-extendl\ 1\ (j + 6)$

**lemma** *tm-PHI1-tm*:

**assumes**  $0 < j$  **and**  $j + 7 < k$  **and**  $G \geq 6$   
**shows** *turing-machine k G (tm-PHI1 j)*  
*<proof>*

**locale** *turing-machine-PHI1* =

**fixes**  $j :: tapeidx$

**begin**

**definition** *tm1*  $\equiv tm-setn (j + 2)\ 1$

**definition**  $tm2 \equiv tm1$  ;; *tm-Psigamma*  $j$   
**definition**  $tm3 \equiv tm2$  ;; *tm-extendl*  $1 (j + 6)$

**lemma** *tm3-eq-tm-PHI1*:  $tm3 = tm-PHI1 j$   
 ⟨*proof*⟩

**context**

**fixes**  $tps0$  :: *tape list* **and**  $k$  *idx*  $H$  :: *nat* **and**  $nss$  :: *nat list list*  
**assumes**  $jk$ :  $length\ tps0 = k$   $1 < j$   $j + 7 < k$   
**and**  $H$ :  $H \geq 3$   
**assumes**  $tps0$ :  
 $tps0 ! 1 = nlltape\ nss$   
 $tps0 ! j = ([0]_N, 1)$   
 $tps0 ! (j + 1) = ([H]_N, 1)$   
 $tps0 ! (j + 2) = ([[]], 1)$   
 $tps0 ! (j + 3) = ([[]], 1)$   
 $tps0 ! (j + 4) = ([[]], 1)$   
 $tps0 ! (j + 5) = ([[]], 1)$   
 $tps0 ! (j + 6) = ([[]], 1)$   
 $tps0 ! (j + 7) = ([[]], 1)$

**begin**

**definition**  $tps1 \equiv tps0$   
 $[j + 2 := ([1]_N, 1)]$

**lemma** *tm1 [transforms-intros]*:  
**assumes**  $ttt = 12$   
**shows** *transforms*  $tm1\ tps0\ ttt\ tps1$   
 ⟨*proof*⟩

**definition**  $tps2 \equiv tps0$   
 $[j + 2 := ([1]_N, 1),$   
 $j + 6 := ([nll-Psi\ 0\ H\ 1]_{NLL}, 1)]$

**lemma** *tm2 [transforms-intros]*:  
**assumes**  $ttt = 12 + 1851 * H^4$   
**shows** *transforms*  $tm2\ tps0\ ttt\ tps2$   
 ⟨*proof*⟩

**definition**  $tps3 \equiv tps0$   
 $[j + 2 := ([1]_N, 1),$   
 $j + 6 := ([nll-Psi\ 0\ H\ 1]_{NLL}, 1),$   
 $1 := nlltape (nss @ nll-Psi\ 0\ H\ 1)]$

**lemma** *tm3*:  
**assumes**  $ttt = 16 + 1851 * H^4 + 2 * nlllength (nll-Psi\ 0\ H\ 1)$   
**shows** *transforms*  $tm3\ tps0\ ttt\ tps3$   
 ⟨*proof*⟩

**lemma** *tm3'*:  
**assumes**  $ttt = 1875 * H^4$   
**shows** *transforms*  $tm3\ tps0\ ttt\ tps3$   
 ⟨*proof*⟩

**end**

**end**

**lemma** *transforms-tm-PHI1*:  
**fixes**  $j$  :: *tapeidx*  
**fixes**  $tps\ tps'$  :: *tape list* **and**  $ttt\ k\ H$  :: *nat* **and**  $nss$  :: *nat list list*  
**assumes**  $length\ tps = k$  **and**  $1 < j$  **and**  $j + 7 < k$  **and**  $H \geq 3$   
**assumes**

```

    tps ! 1 = nlltape nss
    tps ! j = ([0]N, 1)
    tps ! (j + 1) = ([H]N, 1)
    tps ! (j + 2) = ([[]], 1)
    tps ! (j + 3) = ([[]], 1)
    tps ! (j + 4) = ([[]], 1)
    tps ! (j + 5) = ([[]], 1)
    tps ! (j + 6) = ([[]], 1)
    tps ! (j + 7) = ([[]], 1)
assumes tps' = tps
    [j + 2 := ([1]N, 1),
     j + 6 := ([nll-Psi 0 H 1]NLL, 1),
     1 := nlltape (nss @ nll-Psi 0 H 1)]
assumes ttt = 1875 * H ^ 4
shows transforms (tm-PHI1 j) tps ttt tps'
⟨proof⟩

```

### 7.6.3 A Turing machine for $\Phi_2$

The next TM expects a number  $i$  on tape  $j$  and a number  $H$  on tape  $j + 1$ . It appends to the formula on tape 1 the formula  $\Psi([(2i + 1)H, (2i + 2)H], 3) \wedge \Psi([(2i + 2)H, (2i + 3)H], 3)$ .

**definition** *tm-PHI2* :: *tapeidx*  $\Rightarrow$  *machine* **where**

```

tm-PHI2 j  $\equiv$ 
  tm-times2 j ;;
  tm-incr j ;;
  tm-setn (j + 2) 3 ;;
  tm-Psigamma j ;;
  tm-extendl-erase 1 (j + 6) ;;
  tm-incr j ;;
  tm-Psigamma j ;;
  tm-extendl 1 (j + 6)

```

**lemma** *tm-PHI2-tm*:

```

assumes 0 < j and j + 8 < k and G  $\geq$  6
shows turing-machine k G (tm-PHI2 j)
⟨proof⟩

```

**locale** *turing-machine-PHI2* =

**fixes**  $j :: \text{tapeidx}$

**begin**

```

definition tm1  $\equiv$  tm-times2 j
definition tm2  $\equiv$  tm1 ;; tm-incr j
definition tm3  $\equiv$  tm2 ;; tm-setn (j + 2) 3
definition tm4  $\equiv$  tm3 ;; tm-Psigamma j
definition tm5  $\equiv$  tm4 ;; tm-extendl-erase 1 (j + 6)
definition tm7  $\equiv$  tm5 ;; tm-incr j
definition tm8  $\equiv$  tm7 ;; tm-Psigamma j
definition tm9  $\equiv$  tm8 ;; tm-extendl 1 (j + 6)

```

**lemma** *tm9-eq-tm-PHI2*:  $tm9 = tm-PHI2 j$

⟨proof⟩

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $k \text{ idx } H :: \text{nat}$  **and**  $nss :: \text{nat list list}$

**assumes**  $jk$ :  $\text{length } tps0 = k$   $1 < j$   $j + 7 < k$

**and**  $H$ :  $H \geq 3$

**assumes**  $tps0$ :

```

tps0 ! 1 = nlltape nss
tps0 ! j = ([idx]N, 1)
tps0 ! (j + 1) = ([H]N, 1)
tps0 ! (j + 2) = ([[]], 1)
tps0 ! (j + 3) = ([[]], 1)

```

```

    tps0 ! (j + 4) = ([[]], 1)
    tps0 ! (j + 5) = ([[]], 1)
    tps0 ! (j + 6) = ([[]], 1)
    tps0 ! (j + 7) = ([[]], 1)
begin

definition tps1 ≡ tps0
  [j := ([2 * idx]N, 1)]

lemma tm1 [transforms-intros]:
  assumes ttt = 5 + 2 * nlength idx
  shows transforms tm1 tps0 ttt tps1
  ⟨proof⟩

definition tps2 ≡ tps0
  [j := ([2 * idx + 1]N, 1)]

lemma tm2:
  assumes ttt = 10 + 2 * nlength idx + 2 * nlength (2 * idx)
  shows transforms tm2 tps0 ttt tps2
  ⟨proof⟩

lemma tm2' [transforms-intros]:
  assumes ttt = 12 + 4 * nlength idx
  shows transforms tm2 tps0 ttt tps2
  ⟨proof⟩

definition tps3 ≡ tps0
  [j := ([2 * idx + 1]N, 1),
   j + 2 := ([3]N, 1)]

lemma tm3 [transforms-intros]:
  assumes ttt = 26 + 4 * nlength idx
  shows transforms tm3 tps0 ttt tps3
  ⟨proof⟩

definition tps4 ≡ tps0
  [j := ([2 * idx + 1]N, 1),
   j + 2 := ([3]N, 1),
   j + 6 := ([nll-Psi (Suc (2 * idx) * H) H 3]NLL, 1)]

lemma tm4 [transforms-intros]:
  assumes ttt = 26 + 4 * nlength idx + 1851 * H ^ 4 * (nlength (Suc (Suc (2 * idx))))2
  shows transforms tm4 tps0 ttt tps4
  ⟨proof⟩

definition tps5 ≡ tps0
  [j := ([2 * idx + 1]N, 1),
   j + 2 := ([3]N, 1),
   j + 6 := ([[]], 1),
   1 := nlltape (nss @ nll-Psi (H + 2 * idx * H) H 3)]

lemma tm5 [transforms-intros]:
  assumes ttt = 37 + 4 * nlength idx + 1851 * H ^ 4 * (nlength (Suc (Suc (2 * idx))))2 +
    4 * nlllength (nll-Psi (H + 2 * idx * H) H 3)
  shows transforms tm5 tps0 ttt tps5
  ⟨proof⟩

definition tps7 ≡ tps0
  [j := ([2 * idx + 2]N, 1),
   j + 2 := ([3]N, 1),
   j + 6 := ([[]], 1),
   1 := nlltape (nss @ nll-Psi (H + 2 * idx * H) H 3)]

```

**lemma** *tm7*:

**assumes**  $ttt = 42 + 4 * nlength\ idx + 1851 * H^4 * (nlength\ (Suc\ (Suc\ (2 * idx))))^2 +$   
 $4 * nllength\ (nll-Psi\ (H + 2 * idx * H)\ H\ 3) + 2 * nlength\ (Suc\ (2 * idx))$   
**shows** *transforms tm7 tps0 ttt tps7*  
(*proof*)

**lemma** *tm7'* [*transforms-intros*]:

**assumes**  $ttt = 44 + 6 * nlength\ idx + 1851 * H^4 * (nlength\ (Suc\ (Suc\ (2 * idx))))^2 +$   
 $4 * nllength\ (nll-Psi\ (H + 2 * idx * H)\ H\ 3)$   
**shows** *transforms tm7 tps0 ttt tps7*  
(*proof*)

**definition** *tps8*  $\equiv$  *tps0*

$[j := (\lfloor 2 * idx + 2 \rfloor_N, 1),$   
 $j + 2 := (\lfloor 3 \rfloor_N, 1),$   
 $j + 6 := (\lfloor nll-Psi\ (Suc\ (Suc\ (2 * idx)) * H)\ H\ 3 \rfloor_{NLL}, 1),$   
 $1 := nlltape\ (nss\ @\ nll-Psi\ (H + 2 * idx * H)\ H\ 3)]$

**lemma** *tm8* [*transforms-intros*]:

**assumes**  $ttt = 44 + 6 * nlength\ idx + 1851 * H^4 * (nlength\ (Suc\ (Suc\ (2 * idx))))^2 +$   
 $4 * nllength\ (nll-Psi\ (H + 2 * idx * H)\ H\ 3) +$   
 $1851 * H^4 * (nlength\ (Suc\ (Suc\ (Suc\ (2 * idx))))))^2$   
**shows** *transforms tm8 tps0 ttt tps8*  
(*proof*)

**definition** *tps9*  $\equiv$  *tps0*

$[j := (\lfloor 2 * idx + 2 \rfloor_N, 1),$   
 $j + 2 := (\lfloor 3 \rfloor_N, 1),$   
 $j + 6 := (\lfloor nll-Psi\ (Suc\ (Suc\ (2 * idx)) * H)\ H\ 3 \rfloor_{NLL}, 1),$   
 $1 := nlltape\ (nss\ @\ nll-Psi\ (H + 2 * idx * H)\ H\ 3\ @\ nll-Psi\ (2 * H + 2 * idx * H)\ H\ 3)]$

**lemma** *tm9*:

**assumes**  $ttt = 48 + 6 * nlength\ idx + 1851 * H^4 * (nlength\ (Suc\ (Suc\ (2 * idx))))^2 +$   
 $4 * nllength\ (nll-Psi\ (H + 2 * idx * H)\ H\ 3) +$   
 $1851 * H^4 * (nlength\ (Suc\ (Suc\ (Suc\ (2 * idx))))))^2 +$   
 $2 * nllength\ (nll-Psi\ (2 * H + 2 * idx * H)\ H\ 3)$   
**shows** *transforms tm9 tps0 ttt tps9*  
(*proof*)

**lemma** *tm9'*:

**assumes**  $ttt = 3764 * H^4 * (3 + nlength\ (3 * H + 2 * idx * H))^2$   
**shows** *transforms tm9 tps0 ttt tps9*  
(*proof*)

**end**

**end**

**lemma** *transforms-tm-PHI2I*:

**fixes**  $j :: tapeidx$   
**fixes**  $tps\ tps' :: tape\ list$  **and**  $ttt\ k\ idx\ H :: nat$  **and**  $nss :: nat\ list\ list$   
**assumes**  $length\ tps = k$  **and**  $1 < j$  **and**  $j + 8 < k$   
**and**  $H \geq 3$   
**assumes**  
 $tps\ !\ 1 = nlltape\ nss$   
 $tps\ !\ j = (\lfloor idx \rfloor_N, 1)$   
 $tps\ !\ (j + 1) = (\lfloor H \rfloor_N, 1)$   
 $tps\ !\ (j + 2) = (\lfloor [] \rfloor, 1)$   
 $tps\ !\ (j + 3) = (\lfloor [] \rfloor, 1)$   
 $tps\ !\ (j + 4) = (\lfloor [] \rfloor, 1)$   
 $tps\ !\ (j + 5) = (\lfloor [] \rfloor, 1)$   
 $tps\ !\ (j + 6) = (\lfloor [] \rfloor, 1)$

```

    tps ! (j + 7) = ([[]], 1)
    tps ! (j + 8) = ([[]], 1)
assumes ttt = 3764 * H ^ 4 * (3 + nlength (3 * H + 2 * idx * H))^2
assumes tps' = tps
    [j := ([2 * idx + 2]_N, 1),
     j + 2 := ([3]_N, 1),
     j + 6 := ([nll-Psi (Suc (2 * idx)) * H] H 3]_NLL, 1),
     1 := nlltape (nss @ nll-Psi (H + 2 * idx * H) H 3 @ nll-Psi (2 * H + 2 * idx * H) H 3)]
shows transforms (tm-PHI2 j) tps ttt tps'
<proof>

```

## 7.6.4 Turing machines for $\Phi_3$ , $\Phi_4$ , and $\Phi_5$

The CNF formulas  $\Phi_3$ ,  $\Phi_4$ , and  $\Phi_5$  have a similar structure and can thus be handled by the same Turing machine. The following TM has a parameter *step* and the usual tape parameter *j*. It expects on tape *j* a number *idx*, on tape *j* + 1 a number *H*, on tape *j* + 2 a number  $\kappa$ , and on tape *j* + 8 the number  $idx + step \cdot numiter$  for some number *numiter*. It appends to the CNF formula on tape 1 the formula  $\Psi(\gamma_{idx}, \kappa) \wedge \Psi(\gamma_{idx+step \cdot (numiter-1)}, \kappa)$ , where  $\gamma_i = [iH, (i+1)H]$ .

**definition** *tm-PHI345* :: nat  $\Rightarrow$  tapeidx  $\Rightarrow$  machine **where**

```

tm-PHI345 step j  $\equiv$ 
  WHILE tm-equalsn j (j + 8) (j + 3) ;  $\lambda$ rs. rs ! (j + 3) =  $\square$  DO
    tm-Psigamma j ;;
    tm-extendl-erase 1 (j + 6) ;;
    tm-plus-const step j
  DONE

```

**lemma** *tm-PHI345-tm*:

```

assumes G  $\geq$  6 and 0 < j and j + 8 < k
shows turing-machine k G (tm-PHI345 step j)
<proof>

```

**locale** *turing-machine-PHI345* =  
**fixes** step :: nat **and** j :: tapeidx  
**begin**

```

definition tmC  $\equiv$  tm-equalsn j (j + 8) (j + 3)
definition tm1  $\equiv$  tm-Psigamma j
definition tm2  $\equiv$  tm1 ;; tm-extendl-erase 1 (j + 6)
definition tm4  $\equiv$  tm2 ;; tm-plus-const step j
definition tmL  $\equiv$  WHILE tmC ;  $\lambda$ rs. rs ! (j + 3) =  $\square$  DO tm4 DONE

```

**lemma** *tmL-eq-tm-PHI345*: tmL = tm-PHI345 step j  
<proof>

**context**

```

fixes tps0 :: tape list and numiter H k idx kappa :: nat and nss :: nat list list
assumes jk: length tps0 = k 1 < j j + 8 < k
and H: H  $\geq$  3
and kappa: kappa  $\leq$  H
and step: step > 0
assumes tps0:
  tps0 ! 1 = nlltape nss
  tps0 ! j = ([idx]_N, 1)
  tps0 ! (j + 1) = ([H]_N, 1)
  tps0 ! (j + 2) = ([kappa]_N, 1)
  tps0 ! (j + 3) = ([[]], 1)
  tps0 ! (j + 4) = ([[]], 1)
  tps0 ! (j + 5) = ([[]], 1)
  tps0 ! (j + 6) = ([[]], 1)
  tps0 ! (j + 7) = ([[]], 1)
  tps0 ! (j + 8) = ([idx + step * numiter]_N, 1)

```

**begin**

**definition**  $tpsL :: nat \Rightarrow tape\ list\ where$

$tpsL\ t \equiv tps0$

$[j := (\lfloor idx + step * t \rfloor_N, 1),$

$1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (H * (idx + step * i))\ H\ kappa)\ [0..<t]))]$

**lemma**  $tpsL-eq-tps0$ :  $tpsL\ 0 = tps0$

$\langle proof \rangle$

**definition**  $tpsC :: nat \Rightarrow tape\ list\ where$

$tpsC\ t \equiv tps0$

$[j := (\lfloor idx + step * t \rfloor_N, 1),$

$1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (H * (idx + step * i))\ H\ kappa)\ [0..<t])),$

$j + 3 := (\lfloor t = numiter \rfloor_B, 1)]$

**lemma**  $tmC$ :

**assumes**  $t \leq numiter$

**and**  $ttt = 3 * nlength\ (idx + step * t) + 7$

**shows**  $transforms\ tmC\ (tpsL\ t)\ ttt\ (tpsC\ t)$

$\langle proof \rangle$

**lemma**  $tmC'$  [*transforms-intros*]:

**assumes**  $t \leq numiter$

**and**  $ttt = 3 * nlength\ (idx + step * numiter) + 7$

**shows**  $transforms\ tmC\ (tpsL\ t)\ ttt\ (tpsC\ t)$

$\langle proof \rangle$

**definition**  $tpsL0 :: nat \Rightarrow tape\ list\ where$

$tpsL0\ t \equiv tps0$

$[j := (\lfloor idx + step * t \rfloor_N, 1),$

$1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (H * (idx + step * i))\ H\ kappa)\ [0..<t]))]$

**lemma**  $tpsL0-eq-tpsC$ :

**assumes**  $t < numiter$

**shows**  $tpsL0\ t = tpsC\ t$

$\langle proof \rangle$

**definition**  $tpsL1 :: nat \Rightarrow tape\ list\ where$

$tpsL1\ t \equiv tps0$

$[j := (\lfloor idx + step * t \rfloor_N, 1),$

$1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (H * (idx + step * i))\ H\ kappa)\ [0..<t])),$

$j + 6 := (\lfloor nll-Psi\ ((idx+step*t) * H)\ H\ kappa \rfloor_{NLL}, 1)]$

**lemma**  $tm1$  [*transforms-intros*]:

**assumes**  $t < numiter$

**and**  $ttt = 1851 * H^4 * (nlength\ (Suc\ (idx+step*t)))^2$

**shows**  $transforms\ tm1\ (tpsL0\ t)\ ttt\ (tpsL1\ t)$

$\langle proof \rangle$

**definition**  $tpsL2 :: nat \Rightarrow tape\ list\ where$

$tpsL2\ t \equiv tps0$

$[j := (\lfloor idx + step * t \rfloor_N, 1),$

$1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (H * (idx + step * i))\ H\ kappa)\ [0..<t])),$

$j + 6 := (\lfloor \square \rfloor, 1),$

$1 := nlltape\ ((nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (H * (idx + step * i))\ H\ kappa)\ [0..<t]))\ @\ nll-Psi\ ((idx+step*t) * H)\ H\ kappa)]$

**lemma**  $tm2$ :

**assumes**  $t < numiter$

**and**  $ttt = 1851 * H^4 * (nlength\ (Suc\ (idx + step * t)))^2 +$

$(11 + 4 * nlllength\ (nll-Psi\ ((idx + step * t) * H)\ H\ kappa))$

**shows**  $transforms\ tm2\ (tpsL0\ t)\ ttt\ (tpsL2\ t)$



**lemma** *tmL*:  
**assumes**  $ttt = \text{Suc } \text{numiter} * (9 + (6 * \text{step} + 1885) * (H \wedge 4 * (\text{nlength } (\text{idx} + \text{step} * \text{numiter})))^2)$   
**and**  $nn = \text{numiter}$   
**shows** *transforms tmL* (*tpsL* 0) *ttt* (*tpsC* *nn*)  
 $\langle \text{proof} \rangle$

**lemma** *tmL'*:  
**assumes**  $ttt = \text{Suc } \text{numiter} * (9 + (6 * \text{step} + 1885) * (H \wedge 4 * (\text{nlength } (\text{idx} + \text{step} * \text{numiter})))^2)$   
**shows** *transforms tmL* *tps0* *ttt* (*tpsC* *numiter*)  
 $\langle \text{proof} \rangle$

**end**

**end**

**lemma** *transforms-tm-PHI345I*:

**fixes**  $j :: \text{tapeidx}$   
**fixes**  $\text{tps } \text{tps}' :: \text{tape list}$  **and**  $ttt \text{ step } \text{numiter } H \text{ k } \text{idx } \text{kappa} :: \text{nat}$  **and**  $\text{nss} :: \text{nat list list}$   
**assumes**  $\text{length } \text{tps} = k$  **and**  $1 < j$  **and**  $j + 8 < k$   
**and**  $H \geq 3$   
**and**  $\text{kappa} \leq H$   
**and**  $\text{step} > 0$   
**assumes**  
 $\text{tps} ! 1 = \text{nlltape } \text{nss}$   
 $\text{tps} ! j = ([\text{idx}]_N, 1)$   
 $\text{tps} ! (j + 1) = ([H]_N, 1)$   
 $\text{tps} ! (j + 2) = ([\text{kappa}]_N, 1)$   
 $\text{tps} ! (j + 3) = ([\square], 1)$   
 $\text{tps} ! (j + 4) = ([\square], 1)$   
 $\text{tps} ! (j + 5) = ([\square], 1)$   
 $\text{tps} ! (j + 6) = ([\square], 1)$   
 $\text{tps} ! (j + 7) = ([\square], 1)$   
 $\text{tps} ! (j + 8) = ([\text{idx} + \text{step} * \text{numiter}]_N, 1)$   
**assumes**  $ttt = \text{Suc } \text{numiter} * (9 + (6 * \text{step} + 1885) * (H \wedge 4 * (\text{nlength } (\text{idx} + \text{step} * \text{numiter})))^2)$   
**assumes**  $\text{tps}' = \text{tps}$   
 $[j := ([\text{idx} + \text{step} * \text{numiter}]_N, 1),$   
 $1 := \text{nlltape } (\text{nss} @ \text{concat } (\text{map } (\lambda i. \text{nll-Psi } (H * (\text{idx} + \text{step} * i)) H \text{kappa}) [0..<\text{numiter}]))],$   
 $j + 3 := ([1]_N, 1)]$   
**shows** *transforms (tm-PHI345 step j) tps ttt tps'*  
 $\langle \text{proof} \rangle$

### 7.6.5 A Turing machine for $\Phi_6$

The next Turing machine expects a symbol sequence  $zs$  on input tape 0, the number 2 on tape  $j$ , and a number  $H$  on tape  $j + 1$ . It appends to the CNF formula on tape 1 the formula  $\bigwedge_{i=0}^{|zs|-1} \Psi(\gamma_{2+2i}, z_i)$ , where  $z_i$  is 2 or 3 if  $zs_i$  is **0** or **1**, respectively.

**definition** *tm-PHI6*  $:: \text{tapeidx} \Rightarrow \text{machine}$  **where**

*tm-PHI6*  $j \equiv$   
**WHILE**  $\square$  ;  $\lambda rs. rs ! 0 \neq \square$  **DO**  
**IF**  $\lambda rs. rs ! 0 = \mathbf{0}$  **THEN**  
 $\text{tm-setn } (j + 2) \ 2$   
**ELSE**  
 $\text{tm-setn } (j + 2) \ 3$   
**ENDIF** ;;  
 $\text{tm-Psigamma } j$  ;;  
 $\text{tm-extendl-erase } 1 \ (j + 6)$  ;;  
 $\text{tm-setn } (j + 2) \ 0$  ;;  
 $\text{tm-right } 0$  ;;  
 $\text{tm-plus-const } 2 \ j$   
**DONE**

**lemma** *tm-PHI6-tm*:

**assumes**  $G \geq 6$  **and**  $0 < j$  **and**  $j + 7 < k$

**shows** *turing-machine*  $k$   $G$  (*tm-PHI6*  $j$ )  
 ⟨*proof*⟩

**locale** *turing-machine-PHI6* =

**fixes**  $j :: \text{tapeid}$

**begin**

**definition**  $tm1 \equiv \text{IF } \lambda rs. rs ! 0 = \mathbf{0} \text{ THEN } tm\text{-setn } (j + 2) \ 2 \ \text{ELSE } tm\text{-setn } (j + 2) \ 3 \ \text{ENDIF}$

**definition**  $tm2 \equiv tm1 \ ; \ ; \ tm\text{-Psi}\gamma \ j$

**definition**  $tm3 \equiv tm2 \ ; \ ; \ tm\text{-extendl-erase } 1 \ (j + 6)$

**definition**  $tm4 \equiv tm3 \ ; \ ; \ tm\text{-setn } (j + 2) \ 0$

**definition**  $tm5 \equiv tm4 \ ; \ ; \ tm\text{-right } 0$

**definition**  $tm6 \equiv tm5 \ ; \ ; \ tm\text{-plus-const } 2 \ j$

**definition**  $tmL \equiv \text{WHILE } [] \ ; \ ; \ \lambda rs. rs ! 0 \neq \square \ \text{DO } tm6 \ \text{DONE}$

**lemma** *tmL-eq-tm-PHI6*:  $tmL = tm\text{-PHI6 } j$

⟨*proof*⟩

**context**

**fixes**  $tps0 :: \text{tape list}$  **and**  $k$   $H :: \text{nat}$  **and**  $zs :: \text{symbol list}$  **and**  $nss :: \text{nat list list}$

**assumes**  $jk$ :  $\text{length } tps0 = k$   $1 < j$   $j + 7 < k$

**and**  $H$ :  $H \geq 3$

**and**  $zs$ : *bit-symbols*  $zs$

**assumes**  $tps0$ :

$tps0 ! 0 = ([zs], 1)$

$tps0 ! 1 = \text{niltape } nss$

$tps0 ! j = ([2]_N, 1)$

$tps0 ! (j + 1) = ([H]_N, 1)$

$tps0 ! (j + 2) = ([0]_N, 1)$

$tps0 ! (j + 3) = ([[]], 1)$

$tps0 ! (j + 4) = ([[]], 1)$

$tps0 ! (j + 5) = ([[]], 1)$

$tps0 ! (j + 6) = ([[]], 1)$

$tps0 ! (j + 7) = ([[]], 1)$

**begin**

**lemma** *H0*:  $H > 0$

⟨*proof*⟩

**lemma** *H-mult*:  $x \leq H * x$   $x \leq x * H$

⟨*proof*⟩

**definition**  $tpsL :: \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsL \ t \equiv tps0$

$[0 := ([zs], \text{Suc } t),$

$j := ([2 + 2 * t]_N, 1),$

$1 := \text{niltape } (nss @ \text{concat } (\text{map } (\lambda i. \text{nil-Psi } (H * (2 + 2 * i)) \ H \ (zs ! i)) \ [0..<t]))]$

**lemma** *tpsL-eq-tps0*:  $tpsL \ 0 = tps0$

⟨*proof*⟩

**definition**  $tpsL1 :: \text{nat} \Rightarrow \text{tape list}$  **where**

$tpsL1 \ t \equiv tps0$

$[0 := ([zs], \text{Suc } t),$

$j := ([2 + 2 * t]_N, 1),$

$j + 2 := ([zs ! t]_N, 1),$

$1 := \text{niltape } (nss @ \text{concat } (\text{map } (\lambda i. \text{nil-Psi } (H * (2 + 2 * i)) \ H \ (zs ! i)) \ [0..<t]))]$

**lemma** *tm1* [*transforms-intros*]:

**assumes**  $ttt = 16$  **and**  $t < \text{length } zs$

**shows** *transforms*  $tm1$  ( $tpsL \ t$ )  $ttt$  ( $tpsL1 \ t$ )

⟨*proof*⟩

**definition**  $tpsL2 :: nat \Rightarrow tape\ list\ where$

$tpsL2\ t \equiv tps0$   
 $[0 := (\lfloor zs \rfloor, Suc\ t),$   
 $j := (\lfloor 2 + 2 * t \rfloor_N, 1),$   
 $j + 2 := (\lfloor zs ! t \rfloor_N, 1),$   
 $j + 6 := (\lfloor nll-Psi\ (2 * H + 2 * t * H)\ H\ (zs ! t) \rfloor_{NLL}, Suc\ 0),$   
 $1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (H * (2 + 2 * i))\ H\ (zs ! i))\ [0..<t]))]$

**lemma**  $tm2$  [transforms-intros]:

**assumes**  $ttt = 16 + 1851 * H \wedge 4 * (nlength\ (Suc\ (Suc\ (Suc\ (2 * t))))^2$   
**and**  $t < length\ zs$   
**shows**  $transforms\ tm2\ (tpsL\ t)\ ttt\ (tpsL2\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL3 :: nat \Rightarrow tape\ list\ where$

$tpsL3\ t \equiv tps0$   
 $[0 := (\lfloor zs \rfloor, Suc\ t),$   
 $j := (\lfloor 2 + 2 * t \rfloor_N, 1),$   
 $j + 2 := (\lfloor zs ! t \rfloor_N, 1),$   
 $j + 6 := (\lfloor \square \rfloor, 1),$   
 $1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (2 * H + H * (2 * i))\ H\ (zs ! i))\ [0..<Suc\ t]))]$

**lemma**  $tm3$  [transforms-intros]:

**assumes**  $ttt = 27 + 1851 * H \wedge 4 * (nlength\ (Suc\ (Suc\ (Suc\ (2 * t))))^2 +$   
 $4 * nlllength\ (nll-Psi\ (2 * H + 2 * t * H)\ H\ (zs ! t))$   
**and**  $t < length\ zs$   
**shows**  $transforms\ tm3\ (tpsL\ t)\ ttt\ (tpsL3\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL4 :: nat \Rightarrow tape\ list\ where$

$tpsL4\ t \equiv tps0$   
 $[0 := (\lfloor zs \rfloor, Suc\ t),$   
 $j := (\lfloor 2 + 2 * t \rfloor_N, 1),$   
 $j + 2 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 6 := (\lfloor \square \rfloor, 1),$   
 $1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (2 * H + H * (2 * i))\ H\ (zs ! i))\ [0..<Suc\ t]))]$

**lemma**  $tm4$  [transforms-intros]:

**assumes**  $ttt = 41 + 1851 * H \wedge 4 * (nlength\ (Suc\ (Suc\ (Suc\ (2 * t))))^2 +$   
 $4 * nlllength\ (nll-Psi\ (2 * H + 2 * t * H)\ H\ (zs ! t))$   
**and**  $t < length\ zs$   
**shows**  $transforms\ tm4\ (tpsL\ t)\ ttt\ (tpsL4\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL5 :: nat \Rightarrow tape\ list\ where$

$tpsL5\ t \equiv tps0$   
 $[0 := (\lfloor zs \rfloor, Suc\ (Suc\ t)),$   
 $j := (\lfloor 2 + 2 * t \rfloor_N, 1),$   
 $j + 2 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 6 := (\lfloor \square \rfloor, 1),$   
 $1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda i. nll-Psi\ (2 * H + H * (2 * i))\ H\ (zs ! i))\ [0..<Suc\ t]))]$

**lemma**  $tm5$  [transforms-intros]:

**assumes**  $ttt = 42 + 1851 * H \wedge 4 * (nlength\ (Suc\ (Suc\ (Suc\ (2 * t))))^2 +$   
 $4 * nlllength\ (nll-Psi\ (2 * H + 2 * t * H)\ H\ (zs ! t))$   
**and**  $t < length\ zs$   
**shows**  $transforms\ tm5\ (tpsL\ t)\ ttt\ (tpsL5\ t)$   
 $\langle proof \rangle$

**definition**  $tpsL6 :: nat \Rightarrow tape\ list\ where$

$tpsL6\ t \equiv tps0$   
 $[0 := (\lfloor zs \rfloor, Suc\ (Suc\ t)),$   
 $j := (\lfloor 2 + 2 * (Suc\ t) \rfloor_N, 1),$

$j + 2 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 6 := (\lfloor \square \rfloor, 1),$   
 $1 := \text{nlltape } (nss \text{ @ concat } (\text{map } (\lambda i. \text{nll-Psi } (2 * H + H * (2 * i)) H (zs ! i)) [0..<Suc t]))]$

**lemma tm6:**

**assumes**  $ttt = 52 + 1851 * H^4 * (\text{nlength } (\text{Suc } (\text{Suc } (\text{Suc } (2 * t))))^2 +$   
 $4 * \text{nlllength } (\text{nll-Psi } (2 * H + 2 * t * H)) H (zs ! t)) +$   
 $4 * \text{nlength } (4 + 2 * t)$   
**and**  $t < \text{length } zs$   
**shows** *transforms tm6 (tpsL t) ttt (tpsL6 t)*  
*<proof>*

**lemma tpsL6-eq-tpsL:**  $\text{tpsL6 } t = \text{tpsL } (\text{Suc } t)$   
*<proof>*

**lemma tm6':**

**assumes**  $ttt = 133648 * H^6 * \text{length } zs^2$   
**and**  $t < \text{length } zs$   
**shows** *transforms tm6 (tpsL t) ttt (tpsL (Suc t))*  
*<proof>*

**lemma tmL:**

**assumes**  $ttt = 133650 * H^6 * \text{length } zs^3 + 1$   
**shows** *transforms tmL (tpsL 0) ttt (tpsL (length zs))*  
*<proof>*

**lemma tmL':**

**assumes**  $ttt = 133650 * H^6 * \text{length } zs^3 + 1$   
**shows** *transforms tmL tps0 ttt (tpsL (length zs))*  
*<proof>*

**end**

**end**

**lemma transforms-tm-PHI6I:**

**fixes**  $j :: \text{tapeidx}$   
**fixes**  $\text{tps } \text{tps}' :: \text{tape list}$  **and**  $ttt \ k \ H :: \text{nat}$  **and**  $zs :: \text{symbol list}$  **and**  $nss :: \text{nat list list}$   
**assumes**  $\text{length } \text{tps} = k$  **and**  $1 < j$  **and**  $j + 7 < k$   
**and**  $H \geq 3$   
**and** *bit-symbols zs*  
**assumes**  
 $\text{tps } ! \ 1 = \text{nlltape } nss$   
 $\text{tps } ! \ 0 = (\lfloor zs \rfloor, 1)$   
 $\text{tps } ! \ j = (\lfloor 2 \rfloor_N, 1)$   
 $\text{tps } ! \ (j + 1) = (\lfloor H \rfloor_N, 1)$   
 $\text{tps } ! \ (j + 2) = (\lfloor 0 \rfloor_N, 1)$   
 $\text{tps } ! \ (j + 3) = (\lfloor \square \rfloor, 1)$   
 $\text{tps } ! \ (j + 4) = (\lfloor \square \rfloor, 1)$   
 $\text{tps } ! \ (j + 5) = (\lfloor \square \rfloor, 1)$   
 $\text{tps } ! \ (j + 6) = (\lfloor \square \rfloor, 1)$   
 $\text{tps } ! \ (j + 7) = (\lfloor \square \rfloor, 1)$   
**assumes**  $\text{tps}' = \text{tps}$   
 $[0 := (\lfloor zs \rfloor, \text{Suc } (\text{length } zs)),$   
 $j := (\lfloor 2 + 2 * \text{length } zs \rfloor_N, 1),$   
 $1 := \text{nlltape } (nss \text{ @ concat } (\text{map } (\lambda i. \text{nll-Psi } (H * (2 + 2 * i)) H (zs ! i)) [0..<\text{length } zs]))]$   
**assumes**  $ttt = 133650 * H^6 * \text{length } zs^3 + 1$   
**shows** *transforms (tm-PHI6 j) tps ttt tps'*  
*<proof>*

### 7.6.6 A Turing machine for $\Phi_7$

The next Turing machine expects a number  $idx$  on tape  $j$ , a number  $H$  on tape  $j + 1$ , and a number  $numiter$  on tape  $j + 6$ . It appends to the CNF formula on tape 1 the formula  $\bigwedge_{t=0}^{numiter} \Upsilon(\gamma_{idx+2t})$  with  $\gamma_i = [iH, (i + 1)H]$ . This equals  $\Phi_7$  if  $idx = 2n + 4$  and  $numiter = p(n)$ .

**definition** *tm-PHI7* :: *tapeidx*  $\Rightarrow$  *machine* **where**

```
tm-PHI7 j  $\equiv$ 
  WHILE [] ;  $\lambda$ rs. rs ! (j + 6)  $\neq$   $\square$  DO
    tm-Upsilongamma j ;;
    tm-extendl-erase 1 (j + 4) ;;
    tm-plus-const 2 j ;;
    tm-decr (j + 6)
  DONE
```

**lemma** *tm-PHI7-tm*:

```
assumes 0 < j and j + 6 < k and 6  $\leq$  G and 2  $\leq$  k
shows turing-machine k G (tm-PHI7 j)
<proof>
```

**locale** *turing-machine-tm-PHI7* =

```
fixes step :: nat and j :: tapeidx
```

**begin**

**definition** *tmL1*  $\equiv$  *tm-Upsilongamma* j

**definition** *tmL2*  $\equiv$  *tmL1* ;; *tm-extendl-erase* 1 (j + 4)

**definition** *tmL4*  $\equiv$  *tmL2* ;; *tm-plus-const* 2 j

**definition** *tmL5*  $\equiv$  *tmL4* ;; *tm-decr* (j + 6)

**definition** *tmL*  $\equiv$  WHILE [] ;  $\lambda$ rs. rs ! (j + 6)  $\neq$   $\square$  DO *tmL5* DONE

**lemma** *tmL-eq-tm-PHI7*: *tmL* = *tm-PHI7* j

<proof>

**context**

```
fixes tps0 :: tape list and numiter H k idx :: nat and nss :: nat list list
```

```
assumes jk: length tps0 = k 1 < j j + 6 < k
```

```
and H: H  $\geq$  3
```

```
assumes tps0:
```

```
tps0 ! 1 = nlltape nss
tps0 ! j = ([idx]N, 1)
tps0 ! (j + 1) = ([H]N, 1)
tps0 ! (j + 2) = ([[]], 1)
tps0 ! (j + 3) = ([[]], 1)
tps0 ! (j + 4) = ([[]], 1)
tps0 ! (j + 5) = ([[]], 1)
tps0 ! (j + 6) = ([numiter]N, 1)
```

**begin**

**lemma** *nlength-H*: *nlength* H  $\geq$  1

<proof>

**definition** *tpsL* :: nat  $\Rightarrow$  *tape list* **where**

```
tpsL t  $\equiv$  tps0
[j := ([idx + 2 * t]N, 1),
 j + 6 := ([numiter - t]N, 1),
 1 := nlltape (nss @ concat (map ( $\lambda$ t. nll-Upsilon (idx + 2 * t) H) [0..<t]))]
```

**lemma** *tpsL-eq-tps0*: *tpsL* 0 = *tps0*

<proof>

**definition** *tpsL1* :: nat  $\Rightarrow$  *tape list* **where**

```
tpsL1 t  $\equiv$  tps0
[j := ([idx + 2 * t]N, 1),
 j + 6 := ([numiter - t]N, 1),
```

$1 := \text{nlltape } (nss @ \text{concat } (\text{map } (\lambda t. \text{nll-Upsilon } (idx + 2 * t) H) [0..<t])),$   
 $j + 4 := (\lfloor \text{nll-Upsilon } (idx + 2 * t) H \rfloor_{NLL}, 1)$

**lemma** *tmL1* [transforms-intros]:

**assumes**  $t < \text{numiter}$   
**and**  $ttt = 205 * H * (\text{nlength } (idx + 2 * t) + \text{nlength } H)^2$   
**shows** *transforms tmL1* (*tpsL*  $t$ ) *ttt* (*tpsL1*  $t$ )  
*<proof>*

**definition** *tpsL2* ::  $\text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL2 } t \equiv \text{tps0}$   
 $[j := (\lfloor idx + 2 * t \rfloor_N, 1),$   
 $j + 6 := (\lfloor \text{numiter} - t \rfloor_N, 1),$   
 $1 := \text{nlltape } (nss @ \text{concat } (\text{map } (\lambda t. \text{nll-Upsilon } (idx + 2 * t) H) [0..<t]) @ (\text{nll-Upsilon } (idx + 2 * t) H)),$   
 $j + 4 := (\lfloor \lfloor \rfloor, 1)$

**lemma** *tmL2* [transforms-intros]:

**assumes**  $t < \text{numiter}$   
**and**  $ttt = 11 + 205 * H * (\text{nlength } (idx + 2 * t) + \text{nlength } H)^2 +$   
 $4 * \text{nlllength } (\text{nll-Upsilon } (idx + 2 * t) H)$   
**shows** *transforms tmL2* (*tpsL*  $t$ ) *ttt* (*tpsL2*  $t$ )  
*<proof>*

**definition** *tpsL4* ::  $\text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL4 } t \equiv \text{tps0}$   
 $[j := (\lfloor idx + 2 * \text{Suc } t \rfloor_N, 1),$   
 $j + 6 := (\lfloor \text{numiter} - t \rfloor_N, 1),$   
 $1 := \text{nlltape } (nss @ \text{concat } (\text{map } (\lambda t. \text{nll-Upsilon } (idx + 2 * t) H) [0..<t]) @ (\text{nll-Upsilon } (idx + 2 * t) H)),$   
 $j + 4 := (\lfloor \lfloor \rfloor, 1)$

**lemma** *tmL4* [transforms-intros]:

**assumes**  $t < \text{numiter}$   
**and**  $ttt = 21 + 205 * H * (\text{nlength } (idx + 2 * t) + \text{nlength } H)^2 +$   
 $4 * \text{nlllength } (\text{nll-Upsilon } (idx + 2 * t) H) + 4 * \text{nlength } (\text{Suc } (\text{Suc } (idx + 2 * t)))$   
**shows** *transforms tmL4* (*tpsL*  $t$ ) *ttt* (*tpsL4*  $t$ )  
*<proof>*

**definition** *tpsL5* ::  $\text{nat} \Rightarrow \text{tape list}$  **where**

$\text{tpsL5 } t \equiv \text{tps0}$   
 $[j := (\lfloor idx + 2 * \text{Suc } t \rfloor_N, 1),$   
 $j + 6 := (\lfloor \text{numiter} - \text{Suc } t \rfloor_N, 1),$   
 $1 := \text{nlltape } (nss @ \text{concat } (\text{map } (\lambda t. \text{nll-Upsilon } (idx + 2 * t) H) [0..<t]) @ (\text{nll-Upsilon } (idx + 2 * t) H)),$   
 $j + 4 := (\lfloor \lfloor \rfloor, 1)$

**lemma** *tmL5*:

**assumes**  $t < \text{numiter}$   
**and**  $ttt = 29 + 205 * H * (\text{nlength } (idx + 2 * t) + \text{nlength } H)^2 +$   
 $4 * \text{nlllength } (\text{nll-Upsilon } (idx + 2 * t) H) + 4 * \text{nlength } (\text{Suc } (\text{Suc } (idx + 2 * t))) +$   
 $2 * \text{nlength } (\text{numiter} - t)$   
**shows** *transforms tmL5* (*tpsL*  $t$ ) *ttt* (*tpsL5*  $t$ )  
*<proof>*

**lemma** *tpsL5-eq-tpsL*: *tpsL5*  $t = \text{tpsL } (\text{Suc } t)$

*<proof>*

**lemma** *tmL5'* [transforms-intros]:

**assumes**  $t < \text{numiter}$   
**and**  $ttt = 256 * H * (\text{nlength } (idx + 2 * \text{numiter}) + \text{nlength } H)^2$   
**shows** *transforms tmL5* (*tpsL*  $t$ ) *ttt* (*tpsL* (*Suc*  $t$ ))  
*<proof>*

**lemma** *tmL*:

**assumes**  $ttt = \text{numiter} * (257 * H * (\text{nlength } (idx + 2 * \text{numiter}) + \text{nlength } H)^2) + 1$

**shows** *transforms tmL* (tpsL 0) ttt (tpsL numiter)  
 ⟨proof⟩

**lemma** *tmL'*:

**assumes**  $ttt = numiter * 257 * H * (nlength\ idx + 2 * numiter) + nlength\ H)^2 + 1$   
**shows** *transforms tmL* tps0 ttt (tpsL numiter)  
 ⟨proof⟩

**end**

**end**

**lemma** *transforms-tm-PHI7I*:

**fixes** *tps tps'* :: *tape list* **and** *ttt numiter H k idx* :: *nat* **and** *nss* :: *nat list list* **and** *j* :: *tapeidx*  
**assumes**  $length\ tps = k$  **and**  $1 < j$  **and**  $j + 6 < k$

**and**  $H \geq 3$

**assumes**

*tps ! 1 = nlltape nss*

*tps ! j = ([idx]<sub>N</sub>, 1)*

*tps ! (j + 1) = ([H]<sub>N</sub>, 1)*

*tps ! (j + 2) = ([[]], 1)*

*tps ! (j + 3) = ([[]], 1)*

*tps ! (j + 4) = ([[]], 1)*

*tps ! (j + 5) = ([[]], 1)*

*tps ! (j + 6) = ([numiter]<sub>N</sub>, 1)*

**assumes**  $ttt = numiter * 257 * H * (nlength\ idx + 2 * numiter) + nlength\ H)^2 + 1$

**assumes** *tps' = tps*

$j := ([idx + 2 * numiter]<sub>N</sub>, 1)$ ,

$j + 6 := ([0]<sub>N</sub>, 1)$ ,

$1 := nlltape\ (nss\ @\ concat\ (map\ (\lambda t.\ nll-Upsilon\ (idx + 2 * t)\ H)\ [0..<numiter]))$

**shows** *transforms (tm-PHI7 j) tps ttt tps'*

⟨proof⟩

## 7.6.7 A Turing machine for $\Phi_8$

The next TM expects a number *idx* on tape *j* and a number *H* on tape *j + 1*. It appends to the formula on tape 1 the formula  $\Psi([idx \cdot H, (idx + 1)H], 3)$ .

**definition** *tm-PHI8* :: *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-PHI8 j*  $\equiv$

*tm-setn* (j + 2) 3 ;;

*tm-Psigamma* j ;;

*tm-extendl* 1 (j + 6)

**lemma** *tm-PHI8-tm*:

**assumes**  $0 < j$  **and**  $j + 7 < k$  **and**  $G \geq 6$

**shows** *turing-machine* k G (tm-PHI8 j)

⟨proof⟩

**locale** *turing-machine-PHI8* =

**fixes** *j* :: *tapeidx*

**begin**

**definition** *tm1*  $\equiv$  *tm-setn* (j + 2) 3

**definition** *tm2*  $\equiv$  *tm1* ;; *tm-Psigamma* j

**definition** *tm3*  $\equiv$  *tm2* ;; *tm-extendl* 1 (j + 6)

**lemma** *tm3-eq-tm-PHI8*: *tm3* = *tm-PHI8 j*

⟨proof⟩

**context**

**fixes** *tps0* :: *tape list* **and** *k idx H* :: *nat* **and** *nss* :: *nat list list*

**assumes** *jk*:  $length\ tps0 = k$   $1 < j$   $j + 7 < k$

**and** *H*:  $H \geq 3$

```

assumes tps0:
  tps0 ! 1 = nlltape nss
  tps0 ! j = ([idx]N, 1)
  tps0 ! (j + 1) = ([H]N, 1)
  tps0 ! (j + 2) = ([[]], 1)
  tps0 ! (j + 3) = ([[]], 1)
  tps0 ! (j + 4) = ([[]], 1)
  tps0 ! (j + 5) = ([[]], 1)
  tps0 ! (j + 6) = ([[]], 1)
  tps0 ! (j + 7) = ([[]], 1)
begin

definition tps1 ≡ tps0
  [j + 2 := ([3]N, 1)]

lemma tm1 [transforms-intros]:
  assumes ttt = 14
  shows transforms tm1 tps0 ttt tps1
  ⟨proof⟩

definition tps2 ≡ tps0
  [j + 2 := ([3]N, 1),
   j + 6 := ([nll-Psi (idx * H) H 3]NLL, 1)]

lemma tm2 [transforms-intros]:
  assumes ttt = 14 + 1851 * H ^ 4 * nlength (Suc idx) ^ 2
  shows transforms tm2 tps0 ttt tps2
  ⟨proof⟩

definition tps3 ≡ tps0
  [1 := nlltape (nss @ nll-Psi (idx * H) H 3),
   j + 2 := ([3]N, 1),
   j + 6 := ([nll-Psi (idx * H) H 3]NLL, 1)]

lemma tm3:
  assumes ttt = 18 + 1851 * H ^ 4 * (nlength (Suc idx))2 +
    2 * nlllength (nll-Psi (idx * H) H 3)
  shows transforms tm3 tps0 ttt tps3
  ⟨proof⟩

lemma tm3':
  assumes ttt = 18 + 1861 * H ^ 4 * (nlength (Suc idx))2
  shows transforms tm3 tps0 ttt tps3
  ⟨proof⟩

end

end

lemma transforms-tm-PHI8I:
  fixes j :: tapeidx
  fixes tps tps' :: tape list and ttt k idx H :: nat and nss :: nat list list
  assumes length tps = k and 1 < j and j + 7 < k
  and H ≥ 3
  assumes
    tps ! 1 = nlltape nss
    tps ! j = ([idx]N, 1)
    tps ! (j + 1) = ([H]N, 1)
    tps ! (j + 2) = ([[]], 1)
    tps ! (j + 3) = ([[]], 1)
    tps ! (j + 4) = ([[]], 1)
    tps ! (j + 5) = ([[]], 1)
    tps ! (j + 6) = ([[]], 1)

```

```

    tps ! (j + 7) = ([[]], 1)
assumes tps' = tps
    [1 := nlltape (nss @ nll-Psi (idx * H) H 3),
     j + 2 := ([3]N, 1),
     j + 6 := ([nll-Psi (idx * H) H 3]NLL, 1)]
assumes ttt = 18 + 1861 * H ^ 4 * (nlength (Suc idx))^2
shows transforms (tm-PHI8 j) tps ttt tps'
<proof>

```

### 7.6.8 A Turing machine for $\Phi_9$

The CNF formula  $\Phi_9 = \bigwedge_{t=1}^{T'}$  is the most complicated part of  $\Phi$ . Clearly, the main task here is to generate the formulas  $\chi_t$

#### A Turing machine for $\chi_t$

A lemma that will help with some time bounds:

```

lemma pow2-le-2pow2: z ^ 2 ≤ 2 ^ (2*z) for z :: nat
<proof>

```

The next Turing machine can be used to generate  $\chi_t$ . It expects on tape 1 a CNF formula, on tape  $j_1$  the list of positions of  $M$ 's input tape head, on tape  $j_2$  the list of positions of  $M$ 's output tape head, on tapes  $j_3, \dots, j_3 + 3$  the numbers  $N, G, Z$ , and  $T$ , on tape  $j_3 + 4$  the formula  $\psi$ , on tape  $j_3 + 5$  the formula  $\psi'$ , and finally on tape  $j_3 + 6$  the number  $t$ . The TM appends the formula  $\chi_t$  to the formula on tape 1, which should be thought of as an unfinished version of  $\Phi$ .

The TM first computes  $prev(t)$  using  $tm-prev$  and compares it with  $t$ . Depending on the outcome of this comparison it generates either  $\varrho_t$  or  $\varrho'_t$  by concatenating ranges of numbers generated using  $tm-range$ . Then the TM uses  $tm-relabel$  to compute  $\varrho_t(\psi)$  or  $\varrho'_t(\psi')$ . The result equals  $\chi_t$  and is appended to tape 1. Finally  $t$  is incremented and  $T$  is decremented. This is so the TM can be used inside a while loop that initializes  $T$  with  $T'$  and  $t$  with 1.

**definition**  $tm-chi :: tapeidx \Rightarrow tapeidx \Rightarrow tapeidx \Rightarrow machine$  **where**

```

tm-chi j1 j2 j3 ≡
  tm-prev j2 (j3 + 6) ;;
  tm-equalsn (j3 + 11) (j3 + 6) (j3 + 13) ;;
  tm-decr (j3 + 6) ;;
  tm-mult (j3 + 6) (j3 + 2) (j3 + 7) ;;
  tm-add j3 (j3 + 7) ;;
  tm-range (j3 + 7) (j3 + 2) (j3 + 8) ;;
  tm-extend-erase (j3 + 12) (j3 + 8) ;;
  tm-setn (j3 + 7) 0 ;;
  IF λrs. rs ! (j3 + 13) = □ THEN
    tm-mult (j3 + 11) (j3 + 2) (j3 + 7) ;;
    tm-add j3 (j3 + 7) ;;
    tm-range (j3 + 7) (j3 + 2) (j3 + 8) ;;
    tm-extend-erase (j3 + 12) (j3 + 8) ;;
    tm-setn (j3 + 7) 0
  ELSE
    []
  ENDIF ;;
  tm-incr (j3 + 6) ;;
  tm-mult (j3 + 6) (j3 + 2) (j3 + 7) ;;
  tm-add j3 (j3 + 7) ;;
  tm-range (j3 + 7) (j3 + 2) (j3 + 8) ;;
  tm-extend-erase (j3 + 12) (j3 + 8) ;;
  tm-setn (j3 + 11) 0 ;;
  tm-nth j1 (j3 + 6) (j3 + 11) 4 ;;
  tm-setn (j3 + 7) 0 ;;
  tm-mult (j3 + 11) (j3 + 1) (j3 + 7) ;;
  tm-range (j3 + 7) (j3 + 1) (j3 + 8) ;;
  tm-extend-erase (j3 + 12) (j3 + 8) ;;

```

```

tm-setn (j3 + 7) 0 ;;
tm-erase-cr (j3 + 11) ;;
tm-cr (j3 + 12) ;;
IF λrs. rs ! (j3 + 13) = □ THEN
  tm-relabel (j3 + 4) (j3 + 11)
ELSE
  tm-erase-cr (j3 + 13) ;;
  tm-relabel (j3 + 5) (j3 + 11)
ENDIF ;;
tm-erase-cr (j3 + 12) ;;
tm-extendl-erase 1 (j3 + 11) ;;
tm-incr (j3 + 6) ;;
tm-decr (j3 + 3)

```

**lemma** *tm-chi-tm*:

**assumes**  $0 < j1$  **and**  $j1 < j2$  **and**  $j2 < j3$  **and**  $j3 + 17 < k$  **and**  $G \geq 6$   
**shows** *turing-machine*  $k$   $G$  (*tm-chi*  $j1$   $j2$   $j3$ )  
*<proof>*

**locale** *turing-machine-chi* =  
**fixes**  $j1$   $j2$   $j3$  :: *tapeidx*  
**begin**

**definition**  $tm1 \equiv tm\text{-prev } j2$  ( $j3 + 6$ )  
**definition**  $tm2 \equiv tm1$  ;; *tm-equalsn* ( $j3 + 11$ ) ( $j3 + 6$ ) ( $j3 + 13$ )

**definition**  $tm3 \equiv tm2$  ;; *tm-decr* ( $j3 + 6$ )  
**definition**  $tm4 \equiv tm3$  ;; *tm-mult* ( $j3 + 6$ ) ( $j3 + 2$ ) ( $j3 + 7$ )  
**definition**  $tm5 \equiv tm4$  ;; *tm-add*  $j3$  ( $j3 + 7$ )  
**definition**  $tm6 \equiv tm5$  ;; *tm-range* ( $j3 + 7$ ) ( $j3 + 2$ ) ( $j3 + 8$ )  
**definition**  $tm7 \equiv tm6$  ;; *tm-extend-erase* ( $j3 + 12$ ) ( $j3 + 8$ )  
**definition**  $tm8 \equiv tm7$  ;; *tm-setn* ( $j3 + 7$ ) 0

**definition**  $tmT1 \equiv tm\text{-mult}$  ( $j3 + 11$ ) ( $j3 + 2$ ) ( $j3 + 7$ )  
**definition**  $tmT2 \equiv tmT1$  ;; *tm-add*  $j3$  ( $j3 + 7$ )  
**definition**  $tmT3 \equiv tmT2$  ;; *tm-range* ( $j3 + 7$ ) ( $j3 + 2$ ) ( $j3 + 8$ )  
**definition**  $tmT4 \equiv tmT3$  ;; *tm-extend-erase* ( $j3 + 12$ ) ( $j3 + 8$ )  
**definition**  $tmT5 \equiv tmT4$  ;; *tm-setn* ( $j3 + 7$ ) 0

**definition**  $tm89 \equiv IF$   $\lambda rs. rs ! (j3 + 13) = \square$  *THEN*  $tmT5$  *ELSE* [] *ENDIF*  
**definition**  $tm10 \equiv tm8$  ;;  $tm89$

**definition**  $tm11 \equiv tm10$  ;; *tm-incr* ( $j3 + 6$ )  
**definition**  $tm13 \equiv tm11$  ;; *tm-mult* ( $j3 + 6$ ) ( $j3 + 2$ ) ( $j3 + 7$ )  
**definition**  $tm14 \equiv tm13$  ;; *tm-add*  $j3$  ( $j3 + 7$ )  
**definition**  $tm15 \equiv tm14$  ;; *tm-range* ( $j3 + 7$ ) ( $j3 + 2$ ) ( $j3 + 8$ )  
**definition**  $tm16 \equiv tm15$  ;; *tm-extend-erase* ( $j3 + 12$ ) ( $j3 + 8$ )  
**definition**  $tm17 \equiv tm16$  ;; *tm-setn* ( $j3 + 11$ ) 0  
**definition**  $tm18 \equiv tm17$  ;; *tm-nth*  $j1$  ( $j3 + 6$ ) ( $j3 + 11$ ) 4  
**definition**  $tm19 \equiv tm18$  ;; *tm-setn* ( $j3 + 7$ ) 0  
**definition**  $tm20 \equiv tm19$  ;; *tm-mult* ( $j3 + 11$ ) ( $j3 + 1$ ) ( $j3 + 7$ )  
**definition**  $tm21 \equiv tm20$  ;; *tm-range* ( $j3 + 7$ ) ( $j3 + 1$ ) ( $j3 + 8$ )  
**definition**  $tm22 \equiv tm21$  ;; *tm-extend-erase* ( $j3 + 12$ ) ( $j3 + 8$ )  
**definition**  $tm23 \equiv tm22$  ;; *tm-setn* ( $j3 + 7$ ) 0  
**definition**  $tm24 \equiv tm23$  ;; *tm-erase-cr* ( $j3 + 11$ )  
**definition**  $tm25 \equiv tm24$  ;; *tm-cr* ( $j3 + 12$ )

**definition**  $tmE1 \equiv tm\text{-erase-cr}$  ( $j3 + 13$ )  
**definition**  $tmE2 \equiv tmE1$  ;; *tm-relabel* ( $j3 + 5$ ) ( $j3 + 11$ )  
**definition**  $tmTT1 \equiv tm\text{-relabel}$  ( $j3 + 4$ ) ( $j3 + 11$ )

**definition**  $tm2526 \equiv IF$   $\lambda rs. rs ! (j3 + 13) = \square$  *THEN*  $tmTT1$  *ELSE*  $tmE2$  *ENDIF*  
**definition**  $tm26 \equiv tm25$  ;;  $tm2526$

**definition**  $tm27 \equiv tm26 \;; \; tm\text{-erase-cr} \; (j3 + 12)$   
**definition**  $tm28 \equiv tm27 \;; \; tm\text{-extendl-erase} \; 1 \; (j3 + 11)$   
**definition**  $tm29 \equiv tm28 \;; \; tm\text{-incr} \; (j3 + 6)$   
**definition**  $tm30 \equiv tm29 \;; \; tm\text{-decr} \; (j3 + 3)$

**lemma**  $tm30\text{-eq-}tm\text{-chi}$ :  $tm30 = tm\text{-chi} \; j1 \; j2 \; j3$   
 ⟨proof⟩

**context**

**fixes**  $tps0 \:: \text{tape list}$  **and**  $k \; G \; N \; Z \; T' \; T \; t \:: \text{nat}$  **and**  $hp0 \; hp1 \:: \text{nat list}$  **and**  $\psi \; \psi' \:: \text{formula}$   
**fixes**  $nss \:: \text{nat list list}$   
**assumes**  $jk$ :  $\text{length } tps0 = k \; 1 < j1 \; j1 < j2 \; j2 < j3 \; j3 + 17 < k$   
**and**  $G$ :  $G \geq 3$   
**and**  $Z$ :  $Z = 3 * G$   
**and**  $N$ :  $N \geq 1$   
**and**  $len\text{-}hp0$ :  $\text{length } hp0 = \text{Suc } T'$   
**and**  $hp0$ :  $\forall i < \text{length } hp0. \; hp0 \; ! \; i \leq T'$   
**and**  $len\text{-}hp1$ :  $\text{length } hp1 = \text{Suc } T'$   
**and**  $hp1$ :  $\forall i < \text{length } hp1. \; hp1 \; ! \; i \leq T'$   
**and**  $t$ :  $0 < t \; t \leq T'$   
**and**  $T$ :  $0 < T \; T \leq T'$   
**and**  $T'$ :  $T' < N$   
**and**  $psi$ :  $\text{variables } \psi \subseteq \{..<3*Z+G\} \; \text{fsize } \psi \leq (3*Z+G) * 2 \wedge (3*Z+G) \; \text{length } \psi \leq 2 \wedge (3*Z+G)$   
**and**  $psi'$ :  $\text{variables } \psi' \subseteq \{..<2*Z+G\} \; \text{fsize } \psi' \leq (2*Z+G) * 2 \wedge (2*Z+G) \; \text{length } \psi' \leq 2 \wedge (2*Z+G)$

**assumes**  $tps0$ :

$tps0 \; ! \; 1 = \text{nlltape } nss$   
 $tps0 \; ! \; j1 = ([hp0]_{NL}, 1)$   
 $tps0 \; ! \; j2 = ([hp1]_{NL}, 1)$   
 $tps0 \; ! \; j3 = ([N]_N, 1)$   
 $tps0 \; ! \; (j3 + 1) = ([G]_N, 1)$   
 $tps0 \; ! \; (j3 + 2) = ([Z]_N, 1)$   
 $tps0 \; ! \; (j3 + 3) = ([T]_N, 1)$   
 $tps0 \; ! \; (j3 + 4) = ([\text{formula-n } \psi]_{NLL}, 1)$   
 $tps0 \; ! \; (j3 + 5) = ([\text{formula-n } \psi']_{NLL}, 1)$   
 $tps0 \; ! \; (j3 + 6) = ([t]_N, 1)$   
 $\bigwedge i. \; 6 < i \implies i < 17 \implies tps0 \; ! \; (j3 + i) = ([\square], 1)$

**begin**

**lemma**  $Z\text{-ge-1}$ :  $Z \geq 1$   
 ⟨proof⟩

**lemma**  $Z\text{-ge-9}$ :  $Z \geq 9$   
 ⟨proof⟩

**lemma**  $T'\text{-ge-1}$ :  $T' \geq 1$   
 ⟨proof⟩

**lemma**  $tps0'$ :  $\bigwedge i. \; 1 \leq i \implies i < 11 \implies tps0 \; ! \; (j3 + 6 + i) = ([\square], 1)$   
 ⟨proof⟩

The simplifier turns  $j3 + 6 + 3$  into  $9 + j3$ . The next lemma helps with that.

**lemma**  $tps0\text{-sym}$ :  $\bigwedge i. \; 6 < i \implies i < 17 \implies tps0 \; ! \; (i + j3) = ([\square], 1)$   
 ⟨proof⟩

**lemma**  $previous\text{-}hp1\text{-le}$ :  $previous \; hp1 \; t \leq t$   
 ⟨proof⟩

**definition**  $tps1 \equiv tps0$   
 $[j3 + 11 := ([previous \; hp1 \; t]_N, 1)]$

**lemma**  $tm1$  [ $\text{transforms-intros}$ ]:  
**assumes**  $ttt = 71 + 153 * \text{nlength } hp1 \wedge 3$   
**shows**  $\text{transforms } tm1 \; tps0 \; ttt \; tps1$

*<proof>*

**definition**  $tps2 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous } hp1 \ t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous } hp1 \ t = t \rfloor_B, 1)]$

**lemma**  $tm2$  [*transforms-intros*]:

**assumes**  $ttt = 78 + 153 * nlength \ hp1 \wedge 3 + 3 * nlength \ (\min \ (\text{previous } hp1 \ t) \ t)$   
**shows** *transforms*  $tm2 \ tps0 \ ttt \ tps2$   
*<proof>*

**definition**  $tps3 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous } hp1 \ t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous } hp1 \ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t - 1 \rfloor_N, 1)]$

**lemma**  $tm3$  [*transforms-intros*]:

**assumes**  $ttt = 86 + 153 * nlength \ hp1 \wedge 3 + 3 * nlength \ (\min \ (\text{previous } hp1 \ t) \ t) + 2 * nlength \ t$   
**shows** *transforms*  $tm3 \ tps0 \ ttt \ tps3$   
*<proof>*

**definition**  $tps4 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous } hp1 \ t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous } hp1 \ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor (t - 1) * Z \rfloor_N, 1)]$

**lemma**  $tm4$  [*transforms-intros*]:

**assumes**  $ttt = 90 + 153 * nlength \ hp1 \wedge 3 + 3 * nlength \ (\min \ (\text{previous } hp1 \ t) \ t) + 2 * nlength \ t +$   
 $26 * (nlength \ (t - 1) + nlength \ Z) \wedge 2$   
**shows** *transforms*  $tm4 \ tps0 \ ttt \ tps4$   
*<proof>*

**definition**  $tps5 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous } hp1 \ t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous } hp1 \ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor N + (t - 1) * Z \rfloor_N, 1)]$

**lemma**  $tm5$  [*transforms-intros*]:

**assumes**  $ttt = 100 + 153 * nlength \ hp1 \wedge 3 + 3 * nlength \ (\min \ (\text{previous } hp1 \ t) \ t) + 2 * nlength \ t +$   
 $26 * (nlength \ (t - 1) + nlength \ Z) \wedge 2 + 3 * \max \ (nlength \ N) \ (nlength \ ((t - 1) * Z))$   
**shows** *transforms*  $tm5 \ tps0 \ ttt \ tps5$   
*<proof>*

**definition**  $tps6 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous } hp1 \ t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous } hp1 \ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor N + (t - 1) * Z \rfloor_N, 1),$   
 $j3 + 8 := (\lfloor [N + (t - 1) * Z .. < N + (t - 1) * Z + Z] \rfloor_{NL}, 1)]$

**lemma**  $tm6$  [*transforms-intros*]:

**assumes**  $ttt = 100 + 153 * nlength \ hp1 \wedge 3 + 3 * nlength \ (\min \ (\text{previous } hp1 \ t) \ t) + 2 * nlength \ t +$   
 $26 * (nlength \ (t - 1) + nlength \ Z) \wedge 2 + 3 * \max \ (nlength \ N) \ (nlength \ ((t - 1) * Z)) +$   
 $Suc \ Z * (43 + 9 * nlength \ (N + (t - 1) * Z + Z))$   
**shows** *transforms*  $tm6 \ tps0 \ ttt \ tps6$   
*<proof>*

**definition**  $tps7 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous } hp1 \ t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous } hp1 \ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t - 1 \rfloor_N, 1),$

$j3 + 7 := (\lfloor N + (t - 1) * Z \rfloor_N, 1),$   
 $j3 + 12 := nltape [N + (t - 1) * Z .. < N + (t - 1) * Z + Z]$

**lemma** *tm7* [*transforms-intros*]:

**assumes**  $t = 111 + 153 * nlength\ hp1 \wedge 3 + 3 * nlength\ (\min\ (\text{previous}\ hp1\ t)\ t) + 2 * nlength\ t +$   
 $26 * (nlength\ (t - 1) + nlength\ Z) \wedge 2 + 3 * \max\ (nlength\ N)\ (nlength\ ((t - 1) * Z)) +$   
 $Suc\ Z * (43 + 9 * nlength\ (N + (t - 1) * Z + Z)) +$   
 $4 * nlength\ [N + (t - Suc\ 0) * Z .. < N + (t - Suc\ 0) * Z + Z]$   
**shows** *transforms tm7 tps0 ttt tps7*  
*<proof>*

**definition** *tps8*  $\equiv tps0$

$[j3 + 11 := (\lfloor \text{previous}\ hp1\ t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous}\ hp1\ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor 0 \rfloor_N, 1),$   
 $j3 + 12 := nltape [N + (t - 1) * Z .. < N + (t - 1) * Z + Z]$

**lemma** *tm8*:

**assumes**  $t = 121 + 153 * nlength\ hp1 \wedge 3 + 3 * nlength\ (\min\ (\text{previous}\ hp1\ t)\ t) +$   
 $2 * nlength\ t + 26 * (nlength\ (t - 1) + nlength\ Z) \wedge 2 + 3 * \max\ (nlength\ N)\ (nlength\ ((t - 1) * Z)) +$   
 $Suc\ Z * (43 + 9 * nlength\ (N + (t - 1) * Z + Z)) + 4 * nlength\ [N + (t - 1) * Z .. < N + (t - 1) * Z$   
 $+ Z] +$   
 $2 * nlength\ (N + (t - 1) * Z)$   
**shows** *transforms tm8 tps0 ttt tps8*  
*<proof>*

For the next upper bound we have no scruples replacing  $\log T'$ ,  $\log N$ , and  $\log Z$  by  $T'$ ,  $N$  and  $Z$ , respectively. All values are polynomial in  $n$  ( $Z$  is even a constant), so the overall polynomiality is not in jeopardy.

**lemma** *nlength-le*:

**fixes**  $nmax :: nat$  **and**  $ns :: nat\ list$   
**assumes**  $\forall n \in set\ ns. n \leq nmax$   
**shows**  $nlength\ ns \leq Suc\ nmax * length\ ns$   
*<proof>*

**lemma** *nlength-upt-le*:

**fixes**  $a\ b :: nat$   
**shows**  $nlength\ [a .. < b] \leq Suc\ b * (b - a)$   
*<proof>*

**lemma** *nlength-hp1*:  $nlength\ hp1 \leq Suc\ T' * Suc\ T'$   
*<proof>*

**definition** *ttt8*  $\equiv 168 + 153 * Suc\ T' \wedge 6 + 5 * t + 26 * (t + Z) \wedge 2 + 47 * Z + 15 * Z * (N + t * Z)$

**lemma** *tm8'* [*transforms-intros*]: *transforms tm8 tps0 ttt8 tps8*  
*<proof>*

**definition** *tpsT1*  $\equiv tps0$

$[j3 + 11 := (\lfloor \text{previous}\ hp1\ t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous}\ hp1\ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor \text{previous}\ hp1\ t * Z \rfloor_N, 1),$   
 $j3 + 12 := nltape [N + (t - 1) * Z .. < N + (t - 1) * Z + Z]$

**lemma** *tmT1* [*transforms-intros*]:

**assumes**  $t = 4 + 26 * (nlength\ (\text{previous}\ hp1\ t) + nlength\ Z) \wedge 2$   
**shows** *transforms tmT1 tps8 ttt tpsT1*  
*<proof>*

**definition** *tpsT2*  $\equiv tps0$

$[j3 + 11 := (\lfloor \text{previous}\ hp1\ t \rfloor_N, 1),$

$j\mathfrak{J} + 13 := (\lfloor \text{previous hp1 } t = t \rfloor_B, 1),$   
 $j\mathfrak{J} + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j\mathfrak{J} + 7 := (\lfloor N + \text{previous hp1 } t * Z \rfloor_N, 1),$   
 $j\mathfrak{J} + 12 := \text{nltape } [N + (t - 1) * Z..<N + (t - 1) * Z + Z]$

**lemma**  $tmT2$  [transforms-intros]:

**assumes**  $t\mathfrak{t}t = 14 + 26 * (\text{nlength } (\text{previous hp1 } t) + \text{nlength } Z) \wedge 2 +$   
 $3 * \text{max } (\text{nlength } N) (\text{nlength } (\text{previous hp1 } t * Z))$

**shows**  $\text{transforms } tmT2 \text{ tps8 } t\mathfrak{t}t \text{ tps}T2$

$\langle \text{proof} \rangle$

**definition**  $tpsT3 \equiv tps0$

$[j\mathfrak{J} + 11 := (\lfloor \text{previous hp1 } t \rfloor_N, 1),$   
 $j\mathfrak{J} + 13 := (\lfloor \text{previous hp1 } t = t \rfloor_B, 1),$   
 $j\mathfrak{J} + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j\mathfrak{J} + 7 := (\lfloor N + \text{previous hp1 } t * Z \rfloor_N, 1),$   
 $j\mathfrak{J} + 12 := \text{nltape } [N + (t - 1) * Z..<N + (t - 1) * Z + Z],$   
 $j\mathfrak{J} + 8 := (\lfloor [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \rfloor_{NL}, 1)]$

**lemma**  $tmT3$  [transforms-intros]:

**assumes**  $t\mathfrak{t}t = 14 + 26 * (\text{nlength } (\text{previous hp1 } t) + \text{nlength } Z) \wedge 2 +$   
 $3 * \text{max } (\text{nlength } N) (\text{nlength } (\text{previous hp1 } t * Z)) +$

$\text{Suc } Z * (4\mathfrak{J} + 9 * \text{nlength } (N + \text{previous hp1 } t * Z + Z))$

**shows**  $\text{transforms } tmT3 \text{ tps8 } t\mathfrak{t}t \text{ tps}T3$

$\langle \text{proof} \rangle$

**definition**  $tpsT4 \equiv tps0$

$[j\mathfrak{J} + 11 := (\lfloor \text{previous hp1 } t \rfloor_N, 1),$   
 $j\mathfrak{J} + 13 := (\lfloor \text{previous hp1 } t = t \rfloor_B, 1),$   
 $j\mathfrak{J} + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j\mathfrak{J} + 7 := (\lfloor N + \text{previous hp1 } t * Z \rfloor_N, 1),$   
 $j\mathfrak{J} + 12 := \text{nltape}$   
 $(\lfloor [N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$   
 $[N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \rfloor,$   
 $j\mathfrak{J} + 8 := (\lfloor \square \rfloor, 1)]$

**lemma**  $tmT4$  [transforms-intros]:

**assumes**  $t\mathfrak{t}t = 25 + 26 * (\text{nlength } (\text{previous hp1 } t) + \text{nlength } Z) \wedge 2 +$   
 $3 * \text{max } (\text{nlength } N) (\text{nlength } (\text{previous hp1 } t * Z)) +$

$\text{Suc } Z * (4\mathfrak{J} + 9 * \text{nlength } (N + \text{previous hp1 } t * Z + Z)) +$

$4 * \text{nlength } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z]$

**shows**  $\text{transforms } tmT4 \text{ tps8 } t\mathfrak{t}t \text{ tps}T4$

$\langle \text{proof} \rangle$

**definition**  $tpsT5 \equiv tps0$

$[j\mathfrak{J} + 11 := (\lfloor \text{previous hp1 } t \rfloor_N, 1),$   
 $j\mathfrak{J} + 13 := (\lfloor \text{previous hp1 } t = t \rfloor_B, 1),$   
 $j\mathfrak{J} + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j\mathfrak{J} + 7 := (\lfloor 0 \rfloor_N, 1),$   
 $j\mathfrak{J} + 12 := \text{nltape}$   
 $(\lfloor [N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$   
 $[N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \rfloor,$   
 $j\mathfrak{J} + 8 := (\lfloor \square \rfloor, 1)]$

**lemma**  $tmT5$  [transforms-intros]:

**assumes**  $t\mathfrak{t}t = 35 + 26 * (\text{nlength } (\text{previous hp1 } t) + \text{nlength } Z) \wedge 2 +$   
 $3 * \text{max } (\text{nlength } N) (\text{nlength } (\text{previous hp1 } t * Z)) +$

$\text{Suc } Z * (4\mathfrak{J} + 9 * \text{nlength } (N + \text{previous hp1 } t * Z + Z)) +$

$4 * \text{nlength } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] +$

$2 * \text{nlength } (N + \text{previous hp1 } t * Z)$

**shows**  $\text{transforms } tmT5 \text{ tps8 } t\mathfrak{t}t \text{ tps}T5$

$\langle \text{proof} \rangle$

**definition**  $tps9 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous hp1 } t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous hp1 } t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t - 1 \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor 0 \rfloor_N, 1),$   
 $j3 + 12 := \text{nltape}$   
 $([N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$   
 $(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } [])),$   
 $j3 + 8 := ([[], 1])$

**lemma**  $tm89$  [transforms-intros]:

**assumes**  $ttt = 37 + 26 * (\text{nlength } (\text{previous hp1 } t) + \text{nlength } Z) \wedge 2 +$   
 $3 * \max (\text{nlength } N) (\text{nlength } (\text{previous hp1 } t * Z)) +$   
 $\text{Suc } Z * (43 + 9 * \text{nlength } (N + \text{previous hp1 } t * Z + Z)) +$   
 $4 * \text{nlength } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] +$   
 $2 * \text{nlength } (N + \text{previous hp1 } t * Z)$   
**shows**  $\text{transforms } tm89 \text{ tps8 } ttt \text{ tps9}$   
 $\langle \text{proof} \rangle$

**definition**  $ttt10 \equiv ttt8 + 37 + 26 * (\text{nlength } (\text{previous hp1 } t) + \text{nlength } Z) \wedge 2 +$

$3 * \max (\text{nlength } N) (\text{nlength } (\text{previous hp1 } t * Z)) +$   
 $\text{Suc } Z * (43 + 9 * \text{nlength } (N + \text{previous hp1 } t * Z + Z)) +$   
 $4 * \text{nlength } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] +$   
 $2 * \text{nlength } (N + \text{previous hp1 } t * Z)$

**lemma**  $tm10$  [transforms-intros]:  $\text{transforms } tm10 \text{ tps0 } ttt10 \text{ tps9}$

$\langle \text{proof} \rangle$

**definition**  $tps11 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous hp1 } t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous hp1 } t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor 0 \rfloor_N, 1),$   
 $j3 + 12 := \text{nltape}$   
 $([N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$   
 $(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } [])),$   
 $j3 + 8 := ([[], 1])$

**lemma**  $tm11$  [transforms-intros]:

**assumes**  $ttt = ttt10 + 5 + 2 * \text{nlength } (t - 1)$   
**shows**  $\text{transforms } tm11 \text{ tps0 } ttt \text{ tps11}$   
 $\langle \text{proof} \rangle$

**definition**  $tps13 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous hp1 } t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous hp1 } t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor t * Z \rfloor_N, 1),$   
 $j3 + 12 := \text{nltape}$   
 $([N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$   
 $(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } [])),$   
 $j3 + 8 := ([[], 1])$

**lemma**  $tm13$  [transforms-intros]:

**assumes**  $ttt = ttt10 + 9 + 2 * \text{nlength } (t - 1) + 26 * (\text{nlength } t + \text{nlength } Z) \wedge 2$   
**shows**  $\text{transforms } tm13 \text{ tps0 } ttt \text{ tps13}$   
 $\langle \text{proof} \rangle$

**definition**  $tps14 \equiv tps0$

$[j3 + 11 := (\lfloor \text{previous hp1 } t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor \text{previous hp1 } t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor N + t * Z \rfloor_N, 1),$

$j\mathfrak{z} + 12 := \text{nltape}$   
 $([N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$   
 $(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } [])),$   
 $j\mathfrak{z} + 8 := ([[]], 1)$

**lemma** *tm14* [*transforms-intros*]:

**assumes**  $t\text{tt} = t\text{tt}10 + 19 + 2 * \text{nlength } (t - 1) + 26 * (\text{nlength } t + \text{nlength } Z) ^ 2 +$   
 $3 * \text{max } (\text{nlength } N) (\text{nlength } (t * Z))$   
**shows** *transforms tm14 tps0 ttt tps14*  
*<proof>*

**definition** *tps15*  $\equiv$  *tps0*

$[j\mathfrak{z} + 11 := ([\text{previous hp1 } t]_N, 1),$   
 $j\mathfrak{z} + 13 := ([\text{previous hp1 } t = t]_B, 1),$   
 $j\mathfrak{z} + 6 := ([t]_N, 1),$   
 $j\mathfrak{z} + 7 := ([N + t * Z]_N, 1),$   
 $j\mathfrak{z} + 12 := \text{nltape}$   
 $([N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$   
 $(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } [])),$   
 $j\mathfrak{z} + 8 := ([[N + t * Z..<N + t * Z + Z]]_{NL}, 1)$

**lemma** *tm15* [*transforms-intros*]:

**assumes**  $t\text{tt} = t\text{tt}10 + 19 + 2 * \text{nlength } (t - 1) + 26 * (\text{nlength } t + \text{nlength } Z) ^ 2 +$   
 $3 * \text{max } (\text{nlength } N) (\text{nlength } (t * Z)) + \text{Suc } Z * (4\mathfrak{z} + 9 * \text{nlength } (N + t * Z + Z))$   
**shows** *transforms tm15 tps0 ttt tps15*  
*<proof>*

**definition** *tps16*  $\equiv$  *tps0*

$[j\mathfrak{z} + 11 := ([\text{previous hp1 } t]_N, 1),$   
 $j\mathfrak{z} + 13 := ([\text{previous hp1 } t = t]_B, 1),$   
 $j\mathfrak{z} + 6 := ([t]_N, 1),$   
 $j\mathfrak{z} + 7 := ([N + t * Z]_N, 1),$   
 $j\mathfrak{z} + 12 := \text{nltape}$   
 $([N + (t - \text{Suc } 0) * Z..<N + (t - \text{Suc } 0) * Z + Z] @$   
 $(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } []) @$   
 $[N + t * Z..<N + t * Z + Z]),$   
 $j\mathfrak{z} + 8 := ([[]], 1)$

**lemma** *tm16* [*transforms-intros*]:

**assumes**  $t\text{tt} = t\text{tt}10 + 30 + 2 * \text{nlength } (t - 1) + 26 * (\text{nlength } t + \text{nlength } Z) ^ 2 +$   
 $3 * \text{max } (\text{nlength } N) (\text{nlength } (t * Z)) + \text{Suc } Z * (4\mathfrak{z} + 9 * \text{nlength } (N + t * Z + Z)) +$   
 $4 * \text{nlength } [N + t * Z..<N + t * Z + Z]$   
**shows** *transforms tm16 tps0 ttt tps16*  
*<proof>*

**definition** *tps17*  $\equiv$  *tps0*

$[j\mathfrak{z} + 11 := ([0]_N, 1),$   
 $j\mathfrak{z} + 13 := ([\text{previous hp1 } t = t]_B, 1),$   
 $j\mathfrak{z} + 6 := ([t]_N, 1),$   
 $j\mathfrak{z} + 7 := ([N + t * Z]_N, 1),$   
 $j\mathfrak{z} + 12 := \text{nltape}$   
 $([N + (t - \text{Suc } 0) * Z..<N + (t - \text{Suc } 0) * Z + Z] @$   
 $(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } []) @$   
 $[N + t * Z..<N + t * Z + Z]),$   
 $j\mathfrak{z} + 8 := ([[]], 1)$

**lemma** *tm17* [*transforms-intros*]:

**assumes**  $t\text{tt} = t\text{tt}10 + 40 + 2 * \text{nlength } (t - 1) + 26 * (\text{nlength } t + \text{nlength } Z) ^ 2 +$   
 $3 * \text{max } (\text{nlength } N) (\text{nlength } (t * Z)) + \text{Suc } Z * (4\mathfrak{z} + 9 * \text{nlength } (N + t * Z + Z)) +$   
 $4 * \text{nlength } [N + t * Z..<N + t * Z + Z] + 2 * \text{nlength } (\text{previous hp1 } t)$   
**shows** *transforms tm17 tps0 ttt tps17*  
*<proof>*

**definition**  $tps18 \equiv tps0$

$[j3 + 11 := (\lfloor hp0 ! t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor previous\ hp1\ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor N + t * Z \rfloor_N, 1),$   
 $j3 + 12 := nltape$   
 $(\lfloor N + (t - Suc\ 0) * Z..<N + (t - Suc\ 0) * Z + Z \rfloor @$   
 $(if\ previous\ hp1\ t \neq t\ then\ \lfloor N + previous\ hp1\ t * Z..<N + previous\ hp1\ t * Z + Z \rfloor\ else\ \lfloor \rfloor) @$   
 $\lfloor N + t * Z..<N + t * Z + Z \rfloor),$   
 $j3 + 8 := (\lfloor \rfloor, 1)]$

**lemma**  $tm18$  [transforms-intros]:

**assumes**  $tnt = tnt10 + 66 + 2 * nlength\ (t - 1) + 26 * (nlength\ t + nlength\ Z) ^ 2 +$   
 $3 * max\ (nlength\ N)\ (nlength\ (t * Z)) + Suc\ Z * (43 + 9 * nlength\ (N + t * Z + Z)) +$   
 $4 * nlength\ \lfloor N + t * Z..<N + t * Z + Z \rfloor + 2 * nlength\ (previous\ hp1\ t) +$   
 $21 * (nlength\ hp0)^2$   
**shows** *transforms*  $tm18\ tps0\ tnt\ tps18$   
*<proof>*

**definition**  $tps19 \equiv tps0$

$[j3 + 11 := (\lfloor hp0 ! t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor previous\ hp1\ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor 0 \rfloor_N, 1),$   
 $j3 + 12 := nltape$   
 $(\lfloor N + (t - Suc\ 0) * Z..<N + (t - Suc\ 0) * Z + Z \rfloor @$   
 $(if\ previous\ hp1\ t \neq t\ then\ \lfloor N + previous\ hp1\ t * Z..<N + previous\ hp1\ t * Z + Z \rfloor\ else\ \lfloor \rfloor) @$   
 $\lfloor N + t * Z..<N + t * Z + Z \rfloor),$   
 $j3 + 8 := (\lfloor \rfloor, 1)]$

**lemma**  $tm19$  [transforms-intros]:

**assumes**  $tnt = tnt10 + 76 + 2 * nlength\ (t - 1) + 26 * (nlength\ t + nlength\ Z) ^ 2 +$   
 $3 * max\ (nlength\ N)\ (nlength\ (t * Z)) + Suc\ Z * (43 + 9 * nlength\ (N + t * Z + Z)) +$   
 $4 * nlength\ \lfloor N + t * Z..<N + t * Z + Z \rfloor + 2 * nlength\ (previous\ hp1\ t) +$   
 $21 * (nlength\ hp0)^2 + 2 * nlength\ (N + t * Z)$   
**shows** *transforms*  $tm19\ tps0\ tnt\ tps19$   
*<proof>*

**definition**  $tps20 \equiv tps0$

$[j3 + 11 := (\lfloor hp0 ! t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor previous\ hp1\ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor hp0 ! t * G \rfloor_N, 1),$   
 $j3 + 12 := nltape$   
 $(\lfloor N + (t - Suc\ 0) * Z..<N + (t - Suc\ 0) * Z + Z \rfloor @$   
 $(if\ previous\ hp1\ t \neq t\ then\ \lfloor N + previous\ hp1\ t * Z..<N + previous\ hp1\ t * Z + Z \rfloor\ else\ \lfloor \rfloor) @$   
 $\lfloor N + t * Z..<N + t * Z + Z \rfloor),$   
 $j3 + 8 := (\lfloor \rfloor, 1)]$

**definition**  $tnt20 \equiv tnt10 + 80 + 2 * nlength\ (t - 1) + 26 * (nlength\ t + nlength\ Z) ^ 2 +$   
 $3 * max\ (nlength\ N)\ (nlength\ (t * Z)) + Suc\ Z * (43 + 9 * nlength\ (N + t * Z + Z)) +$   
 $4 * nlength\ \lfloor N + t * Z..<N + t * Z + Z \rfloor + 2 * nlength\ (previous\ hp1\ t) +$   
 $21 * (nlength\ hp0)^2 + 2 * nlength\ (N + t * Z) + 26 * (nlength\ (hp0 ! t) + nlength\ G) ^ 2$

**lemma**  $tm20$  [transforms-intros]: *transforms*  $tm20\ tps0\ tnt20\ tps20$

*<proof>*

**definition**  $tps21 \equiv tps0$

$[j3 + 11 := (\lfloor hp0 ! t \rfloor_N, 1),$   
 $j3 + 13 := (\lfloor previous\ hp1\ t = t \rfloor_B, 1),$   
 $j3 + 6 := (\lfloor t \rfloor_N, 1),$   
 $j3 + 7 := (\lfloor hp0 ! t * G \rfloor_N, 1),$   
 $j3 + 12 := nltape$

$$([N + (t - \text{Suc } 0) * Z..<N + (t - \text{Suc } 0) * Z + Z] @$$

$$(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } [])) @$$

$$[N + t * Z..<N + t * Z + Z]),$$

$$j3 + 8 := ([hp0 ! t * G..<hp0 ! t * G + G]_{NL}, 1)]$$

**lemma** *tm21* [*transforms-intros*]:

**assumes**  $ttt = ttt20 + \text{Suc } G * (43 + 9 * \text{nlength } (hp0 ! t * G + G))$

**shows** *transforms tm21 tps0 ttt tps21*

*<proof>*

**abbreviation**  $\sigma \equiv$

$$[N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$$

$$(\text{if previous hp1 } t \neq t \text{ then } [N + \text{previous hp1 } t * Z..<N + \text{previous hp1 } t * Z + Z] \text{ else } [])) @$$

$$[N + t * Z..<N + t * Z + Z] @$$

$$[hp0 ! t * G..<hp0 ! t * G + G]$$

**definition** *tps22*  $\equiv$  *tps0*

$$[j3 + 11 := ([hp0 ! t]_N, 1),$$

$$j3 + 13 := ([previous hp1 t = t]_B, 1),$$

$$j3 + 6 := ([t]_N, 1),$$

$$j3 + 7 := ([hp0 ! t * G]_N, 1),$$

$$j3 + 12 := \text{nltape } \sigma,$$

$$j3 + 8 := ([[]], 1)]$$

**lemma** *tm22* [*transforms-intros*]:

**assumes**  $ttt = ttt20 + 11 + \text{Suc } G * (43 + 9 * \text{nlength } (hp0 ! t * G + G)) +$   
 $4 * \text{nlength } [hp0 ! t * G..<hp0 ! t * G + G]$

**shows** *transforms tm22 tps0 ttt tps22*

*<proof>*

**definition** *tps23*  $\equiv$  *tps0*

$$[j3 + 11 := ([hp0 ! t]_N, 1),$$

$$j3 + 13 := ([previous hp1 t = t]_B, 1),$$

$$j3 + 6 := ([t]_N, 1),$$

$$j3 + 7 := ([0]_N, 1),$$

$$j3 + 12 := \text{nltape } \sigma,$$

$$j3 + 8 := ([[]], 1)]$$

**lemma** *tm23* [*transforms-intros*]:

**assumes**  $ttt = ttt20 + 21 + \text{Suc } G * (43 + 9 * \text{nlength } (hp0 ! t * G + G)) +$   
 $4 * \text{nlength } [hp0 ! t * G..<hp0 ! t * G + G] + 2 * \text{nlength } (hp0 ! t * G)$

**shows** *transforms tm23 tps0 ttt tps23*

*<proof>*

**definition** *tps24*  $\equiv$  *tps0*

$$[j3 + 11 := ([[]], 1),$$

$$j3 + 13 := ([previous hp1 t = t]_B, 1),$$

$$j3 + 6 := ([t]_N, 1),$$

$$j3 + 7 := ([0]_N, 1),$$

$$j3 + 12 := \text{nltape } \sigma,$$

$$j3 + 8 := ([[]], 1)]$$

**lemma** *tm24* [*transforms-intros*]:

**assumes**  $ttt = ttt20 + 28 + \text{Suc } G * (43 + 9 * \text{nlength } (hp0 ! t * G + G)) +$   
 $4 * \text{nlength } [hp0 ! t * G..<hp0 ! t * G + G] + 2 * \text{nlength } (hp0 ! t * G) +$   
 $2 * \text{nlength } (hp0 ! t)$

**shows** *transforms tm24 tps0 ttt tps24*

*<proof>*

**definition** *tps25*  $\equiv$  *tps0*

$$[j3 + 11 := ([[]], 1),$$

$$j3 + 13 := ([previous hp1 t = t]_B, 1),$$

$$j3 + 6 := ([t]_N, 1),$$

$j\mathfrak{J} + 7 := ([\theta]_N, 1),$   
 $j\mathfrak{J} + 12 := ([\sigma]_{NL}, 1),$   
 $j\mathfrak{J} + 8 := ([\square], 1)$

**lemma** *tm25* [transforms-intros]:

**assumes**  $t\mathfrak{t}t = t\mathfrak{t}t20 + 31 + \text{Suc } G * (43 + 9 * \text{nlength } (hp0 ! t * G + G)) +$   
 $4 * \text{nlength } [hp0 ! t * G..<hp0 ! t * G + G] + 2 * \text{nlength } (hp0 ! t * G) +$   
 $2 * \text{nlength } (hp0 ! t) + \text{nlength } \sigma$   
**shows** *transforms tm25 tps0 ttt tps25*  
 <proof>

**definition** *tpsE1*  $\equiv$  *tps0*

$[j\mathfrak{J} + 11 := ([\square], 1),$   
 $j\mathfrak{J} + 13 := ([\square], 1),$   
 $j\mathfrak{J} + 6 := ([t]_N, 1),$   
 $j\mathfrak{J} + 7 := ([\theta]_N, 1),$   
 $j\mathfrak{J} + 12 := ([\sigma]_{NL}, 1),$   
 $j\mathfrak{J} + 8 := ([\square], 1)$

**lemma** *tmE1*:

**assumes**  $t\mathfrak{t}t = 7 + 2 * \text{nlength } (\text{if previous hp1 } t = t \text{ then } 1 \text{ else } 0)$   
**shows** *transforms tmE1 tps25 ttt tpsE1*  
 <proof>

**lemma** *tmE1'* [transforms-intros]:

**assumes**  $t\mathfrak{t}t = 9$   
**shows** *transforms tmE1' tps25 ttt tpsE1*  
 <proof>

**definition** *tpsE2*  $\equiv$  *tps0*

$[j\mathfrak{J} + 11 := \text{nlltape}' (\text{formula-n } (\text{relabel } \sigma \psi')) 0,$   
 $j\mathfrak{J} + 13 := ([\square], 1),$   
 $j\mathfrak{J} + 6 := ([t]_N, 1),$   
 $j\mathfrak{J} + 7 := ([\theta]_N, 1),$   
 $j\mathfrak{J} + 12 := ([\sigma]_{NL}, 1),$   
 $j\mathfrak{J} + 8 := ([\square], 1)$

**lemma** *tmE2* [transforms-intros]:

**assumes**  $t\mathfrak{t}t = 16 + 108 * (\text{nllength } (\text{formula-n } \psi'))^2 * (3 + (\text{nlength } \sigma)^2)$   
**and** *previous hp1 t = t*  
**shows** *transforms tmE2 tps25 ttt tpsE2*  
 <proof>

**definition** *tpsTT1*  $\equiv$  *tps0*

$[j\mathfrak{J} + 11 := \text{nlltape}' (\text{formula-n } (\text{relabel } \sigma \psi)) 0,$   
 $j\mathfrak{J} + 13 := ([\square], 1),$   
 $j\mathfrak{J} + 6 := ([t]_N, 1),$   
 $j\mathfrak{J} + 7 := ([\theta]_N, 1),$   
 $j\mathfrak{J} + 12 := ([\sigma]_{NL}, 1),$   
 $j\mathfrak{J} + 8 := ([\square], 1)$

**lemma** *tmTT1* [transforms-intros]:

**assumes**  $t\mathfrak{t}t = 7 + 108 * (\text{nllength } (\text{formula-n } \psi))^2 * (3 + (\text{nlength } \sigma)^2)$   
**and** *previous hp1 t  $\neq$  t*  
**shows** *transforms tmTT1 tps25 ttt tpsTT1*  
 <proof>

**definition** *tps26*  $\equiv$  *tps0*

$[j\mathfrak{J} + 11 := \text{nlltape}' (\text{formula-n } (\text{relabel } \sigma (\text{if previous hp1 } t = t \text{ then } \psi' \text{ else } \psi))) 0,$   
 $j\mathfrak{J} + 13 := ([\square], 1),$   
 $j\mathfrak{J} + 6 := ([t]_N, 1),$   
 $j\mathfrak{J} + 7 := ([\theta]_N, 1),$   
 $j\mathfrak{J} + 12 := ([\sigma]_{NL}, 1),$

$j3 + 8 := ([\square], 1)$

**lemma** *nllength-psi*:  $nllength \text{ (formula-} n \ \psi) \leq 24 * Z^2 * 2^{(4*Z)}$   
(*proof*)

**lemma** *nllength-psi'*:  $nllength \text{ (formula-} n \ \psi') \leq 24 * Z^2 * 2^{(4*Z)}$   
(*proof*)

**lemma** *tm2526*:

**assumes**  $ttt = 17 + 108 * (24 * Z^2 * 2^{(4*Z)})^2 * (3 + (nllength \ \sigma)^2)$   
**shows** *transforms tm2526 tps25 ttt tps26*  
(*proof*)

**lemma** *nllength- $\sigma$* :  $nllength \ \sigma \leq 12 * T' * Z^2 + 4 * Z * N$   
(*proof*)

**lemma** *tm2526'* [*transforms-intros*]:

**assumes**  $ttt = 17 + 108 * (24 * Z^2 * 2^{(4*Z)})^2 * (3 + (12 * T' * Z^2 + 4 * Z * N)^2)$   
**shows** *transforms tm2526 tps25 ttt tps26*  
(*proof*)

**lemma** *tm26* [*transforms-intros*]:

**assumes**  $ttt = ttt20 + 31 + Suc \ G * (43 + 9 * nlength \ (hp0 \ ! \ t * G + G)) +$   
 $4 * nlength \ [hp0 \ ! \ t * G..<hp0 \ ! \ t * G + G] + 2 * nlength \ (hp0 \ ! \ t * G) +$   
 $2 * nlength \ (hp0 \ ! \ t) + nllength \ \sigma +$   
 $17 + 108 * (24 * Z^2 * 2^{(4*Z)})^2 * (3 + (12 * T' * Z^2 + 4 * Z * N)^2)$   
**shows** *transforms tm26 tps0 ttt tps26*  
(*proof*)

**definition** *tps27*  $\equiv$  *tps0*

$[j3 + 11 := nlltape \text{ (formula-} n \ \text{(relabel } \sigma \ \text{(if previous } hp1 \ t = t \ \text{then } \psi' \ \text{else } \psi))) \ 0,$   
 $j3 + 13 := ([\square], 1),$   
 $j3 + 6 := ([t]_N, 1),$   
 $j3 + 7 := ([\emptyset]_N, 1),$   
 $j3 + 12 := ([\square], 1),$   
 $j3 + 8 := ([\square], 1)]$

**lemma** *tm27*:

**assumes**  $ttt = ttt20 + 38 + Suc \ G * (43 + 9 * nlength \ (hp0 \ ! \ t * G + G)) +$   
 $4 * nllength \ [hp0 \ ! \ t * G..<hp0 \ ! \ t * G + G] + 2 * nlength \ (hp0 \ ! \ t * G) +$   
 $2 * nlength \ (hp0 \ ! \ t) + 3 * nllength \ \sigma +$   
 $17 + 108 * (24 * Z^2 * 2^{(4*Z)})^2 * (3 + (12 * T' * Z^2 + 4 * Z * N)^2)$   
**shows** *transforms tm27 tps0 ttt tps27*  
(*proof*)

**definition** *tps27'*  $\equiv$  *tps0*

$[j3 + 11 := nlltape' \text{ (formula-} n \ \text{(relabel } \sigma \ \text{(if previous } hp1 \ t = t \ \text{then } \psi' \ \text{else } \psi))) \ 0]$

**lemma** *tps27'*:  $tps27 = tps27'$

(*proof*)

**definition** *ttt27*  $= ttt20 + 38 + Suc \ G * (43 + 9 * nlength \ (hp0 \ ! \ t * G + G)) +$

$4 * nllength \ [hp0 \ ! \ t * G..<hp0 \ ! \ t * G + G] + 2 * nlength \ (hp0 \ ! \ t * G) +$   
 $2 * nlength \ (hp0 \ ! \ t) + 3 * nllength \ \sigma +$   
 $17 + 108 * (24 * Z^2 * 2^{(4*Z)})^2 * (3 + (12 * T' * Z^2 + 4 * Z * N)^2)$

**lemma** *tm27'* [*transforms-intros*]: *transforms tm27 tps0 ttt27 tps27'*

(*proof*)

**definition** *tps28*  $\equiv$  *tps0*

$[1 := nlltape \text{ (nss @ formula-} n \ \text{(relabel } \sigma \ \text{(if previous } hp1 \ t = t \ \text{then } \psi' \ \text{else } \psi))),$   
 $j3 + 11 := ([\square], 1)]$

**lemma** *tm28*:

**assumes**  $t_{tt} = t_{tt27} + (11 + 4 * nllength (formula-n (relabel \sigma (if\ previous\ hp1\ t = t\ then\ \psi' \ else\ \psi))))$   
**shows** *transforms tm28 tps0 ttt tps28*  
(*proof*)

**lemma** *nllength-relabel-chi-t*:

$nllength (formula-n (relabel \sigma (if\ previous\ hp1\ t = t\ then\ \psi' \ else\ \psi))) \leq$   
 $Suc (nllength \sigma) * 24 * Z^2 * 2^{(4*Z)}$   
(**is**  $nllength (formula-n (relabel \sigma \ ?\phi)) \leq -$ )  
(*proof*)

**definition**  $tps28' \equiv tps0$

[ $1 := nlltape (nss @ formula-n (relabel \sigma (if\ previous\ hp1\ t = t\ then\ \psi' \ else\ \psi)))$ ]

**lemma** *tps28'*:  $tps28' = tps28$

(*proof*)

**lemma** *tm28'* [*transforms-intros*]:

**assumes**  $t_{tt} = t_{tt27} + (11 + 4 * (Suc (nllength \sigma) * 24 * Z^2 * 2^{(4*Z)}))$   
**shows** *transforms tm28 tps0 ttt tps28'*  
(*proof*)

**definition**  $tps29 \equiv tps0$

[ $1 := nlltape (nss @ formula-n (relabel \sigma (if\ previous\ hp1\ t = t\ then\ \psi' \ else\ \psi)))$ ,  
 $j3 + 6 := (\lfloor Suc\ t \rfloor_N, 1)$ ]

**lemma** *tm29* [*transforms-intros*]:

**assumes**  $t_{tt} = t_{tt27} + 16 + 4 * (Suc (nllength \sigma) * 24 * Z^2 * 2^{(4*Z)}) + 2 * nlength\ t$   
**shows** *transforms tm29 tps0 ttt tps29*  
(*proof*)

**definition**  $tps30 \equiv tps0$

[ $1 := nlltape (nss @ formula-n (relabel \sigma (if\ previous\ hp1\ t = t\ then\ \psi' \ else\ \psi)))$ ,  
 $j3 + 6 := (\lfloor Suc\ t \rfloor_N, 1)$ ,  
 $j3 + 3 := (\lfloor T - 1 \rfloor_N, 1)$ ]

**lemma** *tm30*:

**assumes**  $t_{tt} = t_{tt27} + 24 + 4 * (Suc (nllength \sigma) * 24 * Z^2 * 2^{(4*Z)}) + 2 * nlength\ t + 2 * nlength\ T$   
**shows** *transforms tm30 tps0 ttt tps30*  
(*proof*)

Some helpers for bounding the running time:

**lemma** *pow4-le-2pow4*:  $z^4 \leq 2^{(4*z)}$  **for**  $z :: nat$   
(*proof*)

**lemma** *pow8-le-2pow8*:  $z^8 \leq 2^{(8*z)}$  **for**  $z :: nat$   
(*proof*)

**lemma** *Z-sq-le*:  $Z^2 \leq 2^{(16*Z)}$   
(*proof*)

**lemma** *time-bound*:

$t_{tt27} + 24 + 4 * (Suc (nllength \sigma) * 24 * Z^2 * 2^{(4*Z)}) + 2 * nlength\ t + 2 * nlength\ T \leq 16114765$   
 $* 2^{(16*Z)} * N^6$   
(*proof*)

**lemma** *tm30'*:

**assumes**  $t_{tt} = 16114765 * 2^{(16*Z)} * N^6$   
**shows** *transforms tm30 tps0 ttt tps30*  
(*proof*)

**end**

end

**lemma** *transforms-tm-chi*:

**fixes**  $j1\ j2\ j3 :: \text{tapeidx}$

**fixes**  $\text{tps}\ \text{tps}' :: \text{tape list}$  **and**  $k\ G\ N\ Z\ T'\ T :: \text{nat}$  **and**  $\text{hp0}\ \text{hp1} :: \text{nat list}$  **and**  $\psi\ \psi' :: \text{formula}$

**fixes**  $\text{nss} :: \text{nat list list}$

**assumes**  $\text{length}\ \text{tps} = k$

**and**  $1 < j1\ j1 < j2\ j2 < j3\ j3 + 17 < k$

**and**  $G \geq 3$

**and**  $Z = 3 * G$

**and**  $N \geq 1$

**and**  $\text{length}\ \text{hp0} = \text{Suc}\ T'$

**and**  $\forall i < \text{length}\ \text{hp0}. \text{hp0} ! i \leq T'$

**and**  $\text{length}\ \text{hp1} = \text{Suc}\ T'$

**and**  $\forall i < \text{length}\ \text{hp1}. \text{hp1} ! i \leq T'$

**and**  $0 < t\ t \leq T'$

**and**  $0 < T\ T \leq T'$

**and**  $T' < N$

**and**  $\text{variables}\ \psi \subseteq \{..<3*Z+G\}$   $\text{fsize}\ \psi \leq (3*Z+G) * 2^{\wedge}(3*Z+G)$   $\text{length}\ \psi \leq 2^{\wedge}(3*Z+G)$

**and**  $\text{variables}\ \psi' \subseteq \{..<2*Z+G\}$   $\text{fsize}\ \psi' \leq (2*Z+G) * 2^{\wedge}(2*Z+G)$   $\text{length}\ \psi' \leq 2^{\wedge}(2*Z+G)$

**assumes**

$\text{tps} ! 1 = \text{nulltape}\ \text{nss}$

$\text{tps} ! j1 = ([\text{hp0}]_{NL}, 1)$

$\text{tps} ! j2 = ([\text{hp1}]_{NL}, 1)$

$\text{tps} ! j3 = ([N]_N, 1)$

$\text{tps} ! (j3 + 1) = ([G]_N, 1)$

$\text{tps} ! (j3 + 2) = ([Z]_N, 1)$

$\text{tps} ! (j3 + 3) = ([T]_N, 1)$

$\text{tps} ! (j3 + 4) = ([\text{formula-n}\ \psi]_{NLL}, 1)$

$\text{tps} ! (j3 + 5) = ([\text{formula-n}\ \psi']_{NLL}, 1)$

$\text{tps} ! (j3 + 6) = ([t]_N, 1)$

$\bigwedge i. 6 < i \implies i < 17 \implies \text{tps} ! (j3 + i) = ([\square], 1)$

**assumes**  $\text{tps}' = \text{tps}$

$[1 := \text{nulltape}\ (\text{nss}\ @\ \text{formula-n}\ (\text{relabel}$

$([N + (t - 1) * Z..<N + (t - 1) * Z + Z] @$

$(\text{if}\ \text{previous}\ \text{hp1}\ t \neq t\ \text{then}\ [N + \text{previous}\ \text{hp1}\ t * Z..<N + \text{previous}\ \text{hp1}\ t * Z + Z] \ \text{else}\ []) @$

$[N + t * Z..<N + t * Z + Z] @$

$[\text{hp0} ! t * G..<\text{hp0} ! t * G + G])$

$(\text{if}\ \text{previous}\ \text{hp1}\ t = t\ \text{then}\ \psi' \ \text{else}\ \psi))$ ),

$j3 + 6 := ([\text{Suc}\ t]_N, 1),$

$j3 + 3 := ([T - 1]_N, 1)]$

**assumes**  $\text{tnt} = 16114765 * 2^{\wedge}(16*Z) * N^{\wedge}6$

**shows** *transforms*  $(\text{tm-chi}\ j1\ j2\ j3)\ \text{tps}\ \text{tnt}\ \text{tps}'$

$\langle \text{proof} \rangle$

## A Turing machine for $\Phi_9$ proper

The formula  $\Phi_9$  is a conjunction of formulas  $\chi_t$ . The TM *tm-chi* decreases the number on tape  $j_3 + 3$ . If this tape is initialized with  $T'$ , then a while loop with *tm-chi* as its body will generate  $\Phi_9$ . The next TM is such a machine:

**definition** *tm-PHI9* :: *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *tapeidx*  $\Rightarrow$  *machine* **where**

*tm-PHI9*  $j1\ j2\ j3 \equiv \text{WHILE } [] ; \lambda \text{rs}. \text{rs} ! (j3 + 3) \neq \square \ \text{DO } \text{tm-chi}\ j1\ j2\ j3 \ \text{DONE}$

**lemma** *tm-PHI9-tm*:

**assumes**  $0 < j1$  **and**  $j1 < j2$  **and**  $j2 < j3$  **and**  $j3 + 17 < k$  **and**  $G \geq 6$

**shows** *turing-machine*  $k\ G\ (\text{tm-PHI9}\ j1\ j2\ j3)$

$\langle \text{proof} \rangle$

**lemma** *map-nth*:

**fixes**  $\text{zs}\ \text{ys}\ f\ n\ i$

**assumes**  $\text{zs} = \text{map}\ f\ [0..<n]$  **and**  $i < \text{length}\ \text{zs}$

**shows**  $\text{zs} ! i = f\ i$

$\langle \text{proof} \rangle$

**lemma** *concat-formula-n*:

*concat* (*map* ( $\lambda t. \text{formula-n } (\varphi t)$ ) [*0..<n*]) = *formula-n* (*concat* (*map* ( $\lambda t. \varphi t$ ) [*0..<n*]))  
 ⟨*proof*⟩

**lemma** *upt-append-upt*:

**assumes**  $a \leq b$  **and**  $b \leq c$   
**shows** [*a..<b*] @ [*b..<c*] = [*a..<c*]  
 ⟨*proof*⟩

The semantics of the TM *tm-PHI9* can be proved inside the locale *reduction-sat-x* because it is a fairly simple TM.

**context** *reduction-sat-x*  
**begin**

The TM *tm-chi* is the first TM whose semantics we transfer into the locale *reduction-sat-x*.

**lemma** *tm-chi*:

**fixes** *tps0* :: *tape list* **and**  $k \ t' \ t$  :: *nat* **and**  $j1 \ j2 \ j3$  :: *tapeidx*  
**fixes** *nss* :: *nat list list*  
**assumes**  $jk: \text{length } tps0 = k \ 1 < j1 \ j1 < j2 \ j2 < j3 \ j3 + 17 < k$   
**and**  $t: 0 < t \ t \leq T'$   
**and**  $T: 0 < t' \ t' \leq T'$   
**assumes**  $hp0 = \text{map } (\lambda t. \text{exc } (\text{zeros } m) \ t <\#\> \ 0) \ [0..<Suc \ T']$   
**assumes**  $hp1 = \text{map } (\lambda t. \text{exc } (\text{zeros } m) \ t <\#\> \ 1) \ [0..<Suc \ T']$   
**assumes** *tps0*:  
 $tps0 \ ! \ 1 = \text{nlltape } nss$   
 $tps0 \ ! \ j1 = ([hp0]_{NL}, \ 1)$   
 $tps0 \ ! \ j2 = ([hp1]_{NL}, \ 1)$   
 $tps0 \ ! \ j3 = ([N]_N, \ 1)$   
 $tps0 \ ! \ (j3 + 1) = ([H]_N, \ 1)$   
 $tps0 \ ! \ (j3 + 2) = ([Z]_N, \ 1)$   
 $tps0 \ ! \ (j3 + 3) = ([t']_N, \ 1)$   
 $tps0 \ ! \ (j3 + 4) = ([\text{formula-n } \psi]_{NLL}, \ 1)$   
 $tps0 \ ! \ (j3 + 5) = ([\text{formula-n } \psi']_{NLL}, \ 1)$   
 $tps0 \ ! \ (j3 + 6) = ([t]_N, \ 1)$   
 $\bigwedge i. \ 6 < i \implies i < 17 \implies tps0 \ ! \ (j3 + i) = ([\ ]_N, \ 1)$   
**assumes**  $tps' = tps0$   
 $[1 := \text{nlltape } (nss \ @ \ \text{formula-n } (\chi \ t)),$   
 $j3 + 6 := ([Suc \ t]_N, \ 1),$   
 $j3 + 3 := ([t' - 1]_N, \ 1)]$   
**assumes**  $ttt = 16114765 * 2^{(16 * Z)} * N \wedge 6$   
**shows** *transforms* (*tm-chi*  $j1 \ j2 \ j3$ ) *tps0* *ttt* *tps'*  
 ⟨*proof*⟩

**lemma** *Z-sq-le*:  $Z \wedge 2 \leq 2^{(16 * Z)}$   
 ⟨*proof*⟩

**lemma** *tm-PHI9* [*transforms-intros*]:

**fixes** *tps0* *tps'* :: *tape list* **and**  $k$  :: *nat* **and**  $j1 \ j2 \ j3$  :: *tapeidx*  
**fixes** *nss* :: *nat list list*  
**assumes**  $jk: \text{length } tps0 = k \ 1 < j1 \ j1 < j2 \ j2 < j3 \ j3 + 17 < k$   
**assumes**  $hp0 = \text{map } (\lambda t. \text{exc } (\text{zeros } m) \ t <\#\> \ 0) \ [0..<Suc \ T']$   
**assumes**  $hp1 = \text{map } (\lambda t. \text{exc } (\text{zeros } m) \ t <\#\> \ 1) \ [0..<Suc \ T']$   
**assumes** *tps0*:  
 $tps0 \ ! \ 1 = \text{nlltape } nss$   
 $tps0 \ ! \ j1 = ([hp0]_{NL}, \ 1)$   
 $tps0 \ ! \ j2 = ([hp1]_{NL}, \ 1)$   
 $tps0 \ ! \ j3 = ([N]_N, \ 1)$   
 $tps0 \ ! \ (j3 + 1) = ([H]_N, \ 1)$   
 $tps0 \ ! \ (j3 + 2) = ([Z]_N, \ 1)$   
 $tps0 \ ! \ (j3 + 3) = ([T']_N, \ 1)$   
 $tps0 \ ! \ (j3 + 4) = ([\text{formula-n } \psi]_{NLL}, \ 1)$   
 $tps0 \ ! \ (j3 + 5) = ([\text{formula-n } \psi']_{NLL}, \ 1)$

```

    tps0 ! (j3 + 6) = ([1]N, 1)
     $\wedge i. 6 < i \implies i < 17 \implies tps0 ! (j3 + i) = ([\ ], 1)$ 
assumes tps': tps' = tps0
    [1 := nlltape (nss @ formula-n  $\Phi_9$ ),
     j3 + 6 := ([Suc T']N, 1),
     j3 + 3 := ([0]N, 1)]
assumes ttt = 16114767 * 2  $\wedge$  (16 * Z) * N  $\wedge$  7
shows transforms (tm-PHI9 j1 j2 j3) tps0 ttt tps'
<proof>

```

**end**

**end**

## Chapter 8

# Turing machines for reducing $\mathcal{NP}$ languages to SAT

```
theory Reduction-TM
imports Sat-TM-CNF Oblivious-2-Tape
begin
```

At long last we are going to create a polynomial-time Turing machine that, for a fixed language  $L \in \mathcal{NP}$ , computes for every string  $x$  a CNF formula  $\Phi$  such that  $x \in L$  iff.  $\Phi$  is satisfiable. This concludes the proof of the Cook-Levin theorem.

The CNF formula  $\Phi$  is a conjunction of formulas  $\Phi_0, \dots, \Phi_9$ , and the previous chapter has provided us with Turing machines *tm-PHI0*, *tm-PHI1*, etc. that are supposed to generate these formulas. But only for  $\Phi_9$  has this been proven yet. So our first task is to transfer the Turing machines *tm-PHI0*,  $\dots$ , *tm-PHI8* into the locale *reduction-sat-x* and show that they really do generate the CNF formulas  $\Phi_0, \dots, \Phi_8$ .

The TMs require certain values on their tapes prior to starting. Therefore we build a Turing machine that computes these values. Then, in a final effort, we combine all these TMs to create this article's biggest Turing machine.

### 8.1 Turing machines for parts of $\Phi$ revisited

In this section we restate the semantic lemmas *transforms-tm-PHI0* etc. of the Turing machines *tm-PHI0* etc. in the context of the locale *reduction-sat-x*. This means that the lemmas now have terms like *formula-n*  $\Phi_0$  in them instead of more complicated expressions. It also means that we more clearly see which values the tapes need to contain initially because they are now expressed in terms of values in the locale, such as  $n$ ,  $p(n)$ , or  $m'$ .

```
context reduction-sat-x
begin
```

```
lemma tm-PHI0 [transforms-intros]:
fixes tps tps' :: tape list and j :: tapeidx and ttt k :: nat
assumes length tps = k and 1 < j and j + 8 < k
assumes
  tps ! 1 = ([[]], 1)
  tps ! j = ([m']N, 1)
  tps ! (j + 1) = ([H]N, 1)
  tps ! (j + 2) = ([[]], 1)
  tps ! (j + 3) = ([[]], 1)
  tps ! (j + 4) = ([[]], 1)
  tps ! (j + 5) = ([[]], 1)
  tps ! (j + 6) = ([[]], 1)
  tps ! (j + 7) = ([[]], 1)
  tps ! (j + 8) = ([[]], 1)
assumes tps' = tps
```

$[j := (\lfloor \text{Suc} (\text{Suc } m') \rfloor_N, 1),$   
 $j + 2 := (\lfloor 0 \rfloor_N, 1),$   
 $j + 6 := (\lfloor \text{nll-Psi} (\text{Suc} (\text{Suc } m') * H) H 0 \rfloor_{NLL}, 1),$   
 $1 := \text{nlltape} (\text{formula-n } \Phi_0)]$   
**assumes**  $ttt = 5627 * H^4 * (3 + \text{nlength} (3 * H + m' * H))^2$   
**shows** *transforms* (tm-PHI0 j) tps ttt tps'  
 <proof>

**lemma** *tm-PHI1* [*transforms-intros*]:

**fixes** tps tps' :: *tape list* **and** j :: *tapeidx* **and** ttt k :: *nat* **and** nss :: *nat list list*  
**assumes**  $\text{length } tps = k$  **and**  $1 < j$  **and**  $j + 7 < k$

**assumes**

$tps ! 1 = \text{nlltape } nss$   
 $tps ! j = (\lfloor 0 \rfloor_N, 1)$   
 $tps ! (j + 1) = (\lfloor H \rfloor_N, 1)$   
 $tps ! (j + 2) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 3) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 4) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 5) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 6) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 7) = (\lfloor [] \rfloor, 1)$

**assumes**  $tps' = tps$

$[j + 2 := (\lfloor 1 \rfloor_N, 1),$   
 $j + 6 := (\lfloor \text{nll-Psi } 0 H 1 \rfloor_{NLL}, 1),$   
 $1 := \text{nlltape} (nss @ \text{formula-n } \Phi_1)]$

**assumes**  $ttt = 1875 * H^4$

**shows** *transforms* (tm-PHI1 j) tps ttt tps'

<proof>

**lemma** *tm-PHI2* [*transforms-intros*]:

**fixes** tps tps' :: *tape list* **and** j :: *tapeidx* **and** ttt k :: *nat* **and** nss :: *nat list list*  
**assumes**  $\text{length } tps = k$  **and**  $1 < j$  **and**  $j + 8 < k$

**assumes**  $idx = n$

**assumes**

$tps ! 1 = \text{nlltape } nss$   
 $tps ! j = (\lfloor idx \rfloor_N, 1)$   
 $tps ! (j + 1) = (\lfloor H \rfloor_N, 1)$   
 $tps ! (j + 2) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 3) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 4) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 5) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 6) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 7) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 8) = (\lfloor [] \rfloor, 1)$

**assumes**  $ttt = 3764 * H^4 * (3 + \text{nlength} (3 * H + 2 * idx * H))^2$

**assumes**  $tps' = tps$

$[j := (\lfloor 2 * idx + 2 \rfloor_N, 1),$   
 $j + 2 := (\lfloor 3 \rfloor_N, 1),$   
 $j + 6 := (\lfloor \text{nll-Psi} (\text{Suc} (\text{Suc} (2 * idx)) * H) H 3 \rfloor_{NLL}, 1),$   
 $1 := \text{nlltape} (nss @ \text{formula-n } \Phi_2)]$

**shows** *transforms* (tm-PHI2 j) tps ttt tps'

<proof>

**lemma** *PHI3-correct*:  $\text{concat} (\text{map} (\lambda i. \text{nll-Psi} (H * (1 + 2 * i)) H 2) [0..<n]) = \text{formula-n } \Phi_3$

<proof>

**lemma** *tm-PHI3*:

**fixes** tps tps' :: *tape list* **and** j :: *tapeidx* **and** ttt k :: *nat* **and** nss :: *nat list list*  
**assumes**  $\text{length } tps = k$  **and**  $1 < j$  **and**  $j + 8 < k$

**assumes**

$tps ! 1 = \text{nlltape } nss$   
 $tps ! j = (\lfloor 1 \rfloor_N, 1)$   
 $tps ! (j + 1) = (\lfloor H \rfloor_N, 1)$

$tps ! (j + 2) = (\lfloor 2 \rfloor_N, 1)$   
 $tps ! (j + 3) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 4) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 5) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 6) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 7) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 8) = (\lfloor 1 + 2 * n \rfloor_N, 1)$   
**assumes**  $ttt = Suc\ n * (9 + 1897 * (H \wedge 4 * (nlength\ (1 + 2 * n))^2))$   
**assumes**  $tps' = tps$   
 $[j := (\lfloor 1 + 2 * n \rfloor_N, 1),$   
 $1 := nlltape\ (nss\ @\ formula-n\ \Phi_3),$   
 $j + 3 := (\lfloor 1 \rfloor_N, 1)]$   
**shows**  $transforms\ (tm-PHI345\ 2\ j)\ tps\ ttt\ tps'$   
 $\langle proof \rangle$

**lemma** *PHI4-correct*:

**assumes**  $idx = 2 * n + 2 + 1$  **and**  $kappa = 2$  **and**  $step = 2$  **and**  $numiter = p\ n$   
**shows**  $concat\ (map\ (\lambda i. nll-Psi\ (H * (idx + step * i))\ H\ kappa)\ [0..<numiter]) = formula-n\ \Phi_4$   
 $\langle proof \rangle$

**lemma** *tm-PHI4*:

**fixes**  $tps\ tps' :: tape\ list$  **and**  $j :: tapeidx$  **and**  $ttt\ step\ k :: nat$  **and**  $nss :: nat\ list\ list$   
**assumes**  $length\ tps = k$  **and**  $1 < j$  **and**  $j + 8 < k$  **assumes**  
 $tps ! 1 = nlltape\ nss$   
 $tps ! j = (\lfloor 2 * n + 2 + 1 \rfloor_N, 1)$   
 $tps ! (j + 1) = (\lfloor H \rfloor_N, 1)$   
 $tps ! (j + 2) = (\lfloor 2 \rfloor_N, 1)$   
 $tps ! (j + 3) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 4) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 5) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 6) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 7) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 8) = (\lfloor 2 * n + 2 + 1 + 2 * p\ n \rfloor_N, 1)$   
**assumes**  $ttt = Suc\ (p\ n) * (9 + 1897 * (H \wedge 4 * (nlength\ (2 * n + 2 + 1 + 2 * p\ n))^2))$   
**assumes**  $tps' = tps$   
 $[j := (\lfloor 2 * n + 2 + 1 + 2 * p\ n \rfloor_N, 1),$   
 $1 := nlltape\ (nss\ @\ formula-n\ \Phi_4),$   
 $j + 3 := (\lfloor 1 \rfloor_N, 1)]$   
**shows**  $transforms\ (tm-PHI345\ 2\ j)\ tps\ ttt\ tps'$   
 $\langle proof \rangle$

**lemma** *PHI5-correct*:

**assumes**  $idx = 2 * n + 2 * p\ n + 3$  **and**  $kappa = 0$  **and**  $step = 1$  **and**  $numiter = T'$   
**shows**  $concat\ (map\ (\lambda i. nll-Psi\ (H * (idx + step * i))\ H\ kappa)\ [0..<numiter]) = formula-n\ \Phi_5$   
 $\langle proof \rangle$

**lemma** *tm-PHI5*:

**fixes**  $tps\ tps' :: tape\ list$  **and**  $j :: tapeidx$  **and**  $ttt\ k :: nat$  **and**  $nss :: nat\ list\ list$   
**assumes**  $length\ tps = k$  **and**  $1 < j$  **and**  $j + 8 < k$   
**assumes**  
 $tps ! 1 = nlltape\ nss$   
 $tps ! j = (\lfloor 2 * n + 2 * p\ n + 3 \rfloor_N, 1)$   
 $tps ! (j + 1) = (\lfloor H \rfloor_N, 1)$   
 $tps ! (j + 2) = (\lfloor 0 \rfloor_N, 1)$   
 $tps ! (j + 3) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 4) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 5) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 6) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 7) = (\lfloor \square \rfloor, 1)$   
 $tps ! (j + 8) = (\lfloor 2 * n + 2 * p\ n + 3 + T' \rfloor_N, 1)$   
**assumes**  $ttt = Suc\ T' * (9 + 1891 * (H \wedge 4 * (nlength\ (2 * n + 2 * p\ n + 3 + T'))^2))$   
**assumes**  $tps' = tps$   
 $[j := (\lfloor 2 * n + 2 * p\ n + 3 + T' \rfloor_N, 1),$

$1 := \text{nlltape } (nss \text{ @ formula-}n \Phi_5),$   
 $j + 3 := (\lfloor 1 \rfloor_N, 1)$   
**shows** *transforms* (tm-PHI345 1 j) tps ttt tps'  
 <proof>

**lemma** *PHI6-correct*:

$\text{concat } (\text{map } (\lambda i. \text{nll-Psi } (H * (2 + 2 * i)) H (xs ! i)) [0..<\text{length } xs]) = \text{formula-}n \Phi_6$   
 <proof>

**lemma** *tm-PHI6* [*transforms-intros*]:

**fixes** tps tps' :: *tape list* **and** j :: *tapeidx* **and** ttt k :: *nat* **and** nss :: *nat list list*  
**assumes**  $\text{length } tps = k$  **and**  $1 < j$  **and**  $j + 7 < k$

**assumes**

$tps ! 1 = \text{nlltape } nss$   
 $tps ! 0 = (\lfloor xs \rfloor, 1)$   
 $tps ! j = (\lfloor 2 \rfloor_N, 1)$   
 $tps ! (j + 1) = (\lfloor H \rfloor_N, 1)$   
 $tps ! (j + 2) = (\lfloor 0 \rfloor_N, 1)$   
 $tps ! (j + 3) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 4) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 5) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 6) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 7) = (\lfloor [] \rfloor, 1)$

**assumes**  $tps' = tps$

$[0 := (\lfloor xs \rfloor, \text{Suc } n),$   
 $j := (\lfloor 2 + 2 * n \rfloor_N, 1),$   
 $1 := \text{nlltape } (nss \text{ @ formula-}n \Phi_6)]$

**assumes**  $\text{ttt} = 133650 * H \wedge 6 * n \wedge 3 + 1$

**shows** *transforms* (tm-PHI6 j) tps ttt tps'

<proof>

**lemma** *PHI7-correct*:

**assumes**  $\text{idx} = 2 * n + 4$  **and**  $\text{numiter} = p * n$

**shows**  $\text{concat } (\text{map } (\lambda i. \text{nll-Upsilon } (\text{idx} + 2 * i) H) [0..<\text{numiter}]) = \text{formula-}n \Phi_7$   
 <proof>

**lemma** *tm-PHI7* [*transforms-intros*]:

**fixes** tps tps' :: *tape list* **and** j :: *tapeidx* **and** ttt *numiter* k *idx* :: *nat* **and** nss :: *nat list list*  
**assumes**  $\text{length } tps = k$  **and**  $1 < j$  **and**  $j + 6 < k$

**assumes**

$tps ! 1 = \text{nlltape } nss$   
 $tps ! j = (\lfloor 2 * n + 4 \rfloor_N, 1)$   
 $tps ! (j + 1) = (\lfloor H \rfloor_N, 1)$   
 $tps ! (j + 2) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 3) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 4) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 5) = (\lfloor [] \rfloor, 1)$   
 $tps ! (j + 6) = (\lfloor p * n \rfloor_N, 1)$

**assumes**  $\text{ttt} = p * n * 257 * H * (\text{nlength } (2 * n + 4 + 2 * p * n) + \text{nlength } H)^2 + 1$

**assumes**  $tps' = tps$

$[j := (\lfloor 2 * n + 4 + 2 * p * n \rfloor_N, 1),$   
 $j + 6 := (\lfloor 0 \rfloor_N, 1),$   
 $1 := \text{nlltape } (nss \text{ @ formula-}n \Phi_7)]$

**shows** *transforms* (tm-PHI7 j) tps ttt tps'

<proof>

**lemma** *tm-PHI8* [*transforms-intros*]:

**fixes** tps tps' :: *tape list* **and** j :: *tapeidx* **and** ttt k *idx* :: *nat* **and** nss :: *nat list list*  
**assumes**  $\text{length } tps = k$  **and**  $1 < j$  **and**  $j + 7 < k$

**assumes**  $\text{idx} = 1 + 3 * T' + m'$

**assumes**

$tps ! 1 = \text{nlltape } nss$   
 $tps ! j = (\lfloor 1 + 3 * T' + m' \rfloor_N, 1)$

```

    tps ! (j + 1) = ([H]N, 1)
    tps ! (j + 2) = ([[]], 1)
    tps ! (j + 3) = ([[]], 1)
    tps ! (j + 4) = ([[]], 1)
    tps ! (j + 5) = ([[]], 1)
    tps ! (j + 6) = ([[]], 1)
    tps ! (j + 7) = ([[]], 1)
  assumes tps' = tps
    [1 := nlltape (nss @ formula-n Φ8),
     j + 2 := ([β]N, 1),
     j + 6 := ([formula-n Φ8]NLL, 1)]
  assumes ttt = 18 + 1861 * H ^ 4 * (nlength (Suc (1 + 3 * T' + m')))2
  shows transforms (tm-PHI8 j) tps ttt tps'
<proof>

end

```

## 8.2 A Turing machine for initialization

As we have seen in the previous section, the Turing machines *tm-PHI0* etc. expect some tapes to contain certain values that depend on the verifier TM  $M$ . In this section we construct the TM *tm-PHI-init* that computes these values.

The TM expects the string  $x$  on the input tape. Then it determines the length  $n$  of  $x$  and stores it on tape 11. Then it computes the value  $p(n)$  and stores it on tape 15. Then it computes  $m = 2n + 2p(n) + 2$  and stores it on tape 16. It then writes  $0^m$  to tape 9 and runs *tm-list-headpos*, which writes the sequences of head positions for the input and work/output tape of the verifier TM  $M$  to tapes 4 and 7, respectively. The length of these lists determines  $T'$ , which is written to tape 17. From this and  $m$  the TM computes  $m'$  and writes it to tape 18. It then writes  $H$ , which is hard-coded, to tape 19 and finally  $N = H \cdot m'$  to tape 20.

We assume that the TM starts in a configuration where the input tape head and the heads on tapes with index greater than 10 are positioned on cell number 1, whereas all other tapes are on cell number 0 as usual. The TM has no tape parameters, as all tapes are fixed to work with the final TM later.

As with other TMs before, we will define and analyze the TM on the theory level and then transfer the semantics to the locale *reduction-sat-x*.

**definition** *tm-PHI-init* ::  $nat \Rightarrow machine \Rightarrow (nat \Rightarrow nat) \Rightarrow machine$  **where**

```

tm-PHI-init G M p ≡
  tm-right 9 ;;
  tm-length-input 11 ;;
  tm-polynomial p 11 ;;
  tm-copyn 15 16 ;;
  tm-add 11 16 ;;
  tm-incr 16 ;;
  tm-times2 16 ;;
  tm-copyn 16 17 ;;
  tm-write-replicate 2 17 9 ;;
  tm-left 9 ;;
  tm-list-headpos G M 2 ;;
  tm-count 4 17 4 ;;
  tm-decr 17 ;;
  tm-copyn 16 18 ;;
  tm-incr 18 ;;
  tm-add 17 18 ;;
  tm-setn 19 (max G (length M)) ;;
  tm-mult 18 19 20

```

**lemma** *tm-PHI-init-tm*:

```

fixes H k
assumes turing-machine 2 G M and k > 20 and H ≥ Suc (length M) and H ≥ G
assumes H ≥ 5
shows turing-machine k H (tm-PHI-init G M p)

```

*<proof>*

```
locale turing-machine-PHI-init =
  fixes G :: nat and M :: machine and p :: nat ⇒ nat
begin

definition tm3 ≡ tm-right 9
definition tm4 ≡ tm3 ;; tm-length-input 11
definition tm5 ≡ tm4 ;; tm-polynomial p 11
definition tm6 ≡ tm5 ;; tm-copyn 15 16
definition tm7 ≡ tm6 ;; tm-add 11 16
definition tm8 ≡ tm7 ;; tm-incr 16
definition tm9 ≡ tm8 ;; tm-times2 16
definition tm10 ≡ tm9 ;; tm-copyn 16 17
definition tm11 ≡ tm10 ;; tm-write-replicate 2 17 9
definition tm12 ≡ tm11 ;; tm-left 9
definition tm13 ≡ tm12 ;; tm-list-headpos G M 2
definition tm14 ≡ tm13 ;; tm-count 4 17 4
definition tm15 ≡ tm14 ;; tm-decr 17
definition tm16 ≡ tm15 ;; tm-copyn 16 18
definition tm17 ≡ tm16 ;; tm-incr 18
definition tm18 ≡ tm17 ;; tm-add 17 18
definition tm19 ≡ tm18 ;; tm-setn 19 (max G (length M))
definition tm20 ≡ tm19 ;; tm-mult 18 19 20
```

```
lemma tm20-eq-tm-PHI-init: tm20 = tm-PHI-init G M p
  <proof>
```

**context**

```
fixes k H thalt :: nat and tps0 :: tape list and xs zs :: symbol list
assumes poly-p: polynomial p
  and M-tm: turing-machine 2 G M
  and k: k = length tps0 20 < k
  and H: H = max G (length M)
  and xs: bit-symbols xs
  and zs: zs = 2 # 2 # replicate (2 * length xs + 2 * p (length xs)) 2
assumes thalt:
  ∀ t < thalt. fst (execute M (start-config 2 zs) t) < length M
  fst (execute M (start-config 2 zs) thalt) = length M
assumes tps0:
  tps0 ! 0 = ([xs], 1)
  ∧ i. 0 < i ⇒ i ≤ 10 ⇒ tps0 ! i = ([ ], 0)
  ∧ i. 10 < i ⇒ i < k ⇒ tps0 ! i = ([ ], 1)
```

**begin**

```
lemma G: G ≥ 4
  <proof>
```

```
lemma H: H ≥ length M H ≥ G
  <proof>
```

```
definition tps3 ≡ tps0
  [9 := ([ ], 1)]
```

```
lemma tm3 [transforms-intros]: transforms tm3 tps0 1 tps3
  <proof>
```

**abbreviation** n ≡ length xs

```
definition tps4 ≡ tps0
  [9 := ([ ], 1),
  11 := ([n]N, 1)]
```

**lemma** *tm4* [*transforms-intros*]:  
**assumes**  $ttt = 5 + 11 * (\text{length } xs)^2$   
**shows** *transforms tm4 tps0 ttt tps4*  
*<proof>*

**definition** *tps5*  $\equiv$  *tps0*  
 $[9 := ([[]], 1),$   
 $11 := ([n]_N, 1),$   
 $15 := ([p\ n]_N, 1)]$

**lemma** *tm5* [*transforms-intros*]:  
**assumes**  $ttt = 5 + 11 * (\text{length } xs)^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength (\text{length } xs))^2)$   
**shows** *transforms tm5 tps0 ttt tps5*  
*<proof>*

**definition** *tps6*  $\equiv$  *tps0*  
 $[9 := ([[]], 1),$   
 $11 := ([n]_N, 1),$   
 $15 := ([p\ n]_N, 1),$   
 $16 := ([p\ n]_N, 1)]$

**lemma** *tm6* [*transforms-intros*]:  
**assumes**  $ttt = 19 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) + 3 * nlength\ (p\ n)$   
**shows** *transforms tm6 tps0 ttt tps6*  
*<proof>*

**definition** *tps7*  $\equiv$  *tps0*  
 $[9 := ([[]], 1),$   
 $11 := ([n]_N, 1),$   
 $15 := ([p\ n]_N, 1),$   
 $16 := ([n + p\ n]_N, 1)]$

**lemma** *tm7* [*transforms-intros*]:  
**assumes**  $ttt = 29 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * \max\ (nlength\ n)\ (nlength\ (p\ n))$   
**shows** *transforms tm7 tps0 ttt tps7*  
*<proof>*

**definition** *tps8*  $\equiv$  *tps0*  
 $[9 := ([[]], 1),$   
 $11 := ([n]_N, 1),$   
 $15 := ([p\ n]_N, 1),$   
 $16 := ([Suc\ (n + p\ n)]_N, 1)]$

**lemma** *tm8* [*transforms-intros*]:  
**assumes**  $ttt = 34 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * \max\ (nlength\ n)\ (nlength\ (p\ n)) + 2 * nlength\ (n + p\ n)$   
**shows** *transforms tm8 tps0 ttt tps8*  
*<proof>*

**definition** *tps9*  $\equiv$  *tps0*  
 $[9 := ([[]], 1),$   
 $11 := ([n]_N, 1),$   
 $15 := ([p\ n]_N, 1),$   
 $16 := ([2 * Suc\ (n + p\ n)]_N, 1)]$

**lemma** *tm9* [*transforms-intros*]:  
**assumes**  $ttt = 39 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * \max\ (nlength\ n)\ (nlength\ (p\ n)) + 2 * nlength\ (n + p\ n) +$   
 $2 * nlength\ (Suc\ (n + p\ n))$   
**shows** *transforms tm9 tps0 ttt tps9*  
*<proof>*

**definition**  $tps10 \equiv tps0$

[9 := ( $\lfloor \square \rfloor$ , 1),  
 11 := ( $\lfloor n \rfloor_N$ , 1),  
 15 := ( $\lfloor p \ n \rfloor_N$ , 1),  
 16 := ( $\lfloor 2 * Suc (n + p \ n) \rfloor_N$ , 1),  
 17 := ( $\lfloor 2 * Suc (n + p \ n) \rfloor_N$ , 1)]

**lemma**  $tm10$  [transforms-intros]:

**assumes**  $ttt = 53 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength \ n)^2) +$   
 $3 * nlength (p \ n) + 3 * \max (nlength \ n) (nlength (p \ n)) + 2 * nlength (n + p \ n) +$   
 $2 * nlength (Suc (n + p \ n)) + 3 * nlength (Suc (Suc (2 * n + 2 * p \ n)))$

**shows**  $transforms \ tm10 \ tps0 \ ttt \ tps10$

$\langle proof \rangle$

**definition**  $tps11 \equiv tps0$

[9 := ( $\lfloor zs \rfloor$ , 1),  
 11 := ( $\lfloor n \rfloor_N$ , 1),  
 15 := ( $\lfloor p \ n \rfloor_N$ , 1),  
 16 := ( $\lfloor 2 * Suc (n + p \ n) \rfloor_N$ , 1),  
 17 := ( $\lfloor 0 \rfloor_N$ , 1)]

**lemma**  $tm11$  [transforms-intros]:

**assumes**  $ttt = 57 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength \ n)^2) +$   
 $3 * nlength (p \ n) + 3 * \max (nlength \ n) (nlength (p \ n)) + 2 * nlength (n + p \ n) +$   
 $2 * nlength (Suc (n + p \ n)) + 3 * nlength (Suc (Suc (2 * n + 2 * p \ n))) +$   
 $Suc (Suc (2 * n + 2 * p \ n)) * (12 + 2 * nlength (Suc (Suc (2 * n + 2 * p \ n))))$

**shows**  $transforms \ tm11 \ tps0 \ ttt \ tps11$

$\langle proof \rangle$

**definition**  $tps12 \equiv tps0$

[9 := ( $\lfloor zs \rfloor$ , 0),  
 11 := ( $\lfloor n \rfloor_N$ , 1),  
 15 := ( $\lfloor p \ n \rfloor_N$ , 1),  
 16 := ( $\lfloor 2 * Suc (n + p \ n) \rfloor_N$ , 1),  
 17 := ( $\lfloor 0 \rfloor_N$ , 1)]

**lemma**  $tm12$  [transforms-intros]:

**assumes**  $ttt = 82 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength \ n)^2) +$   
 $3 * nlength (p \ n) + 3 * \max (nlength \ n) (nlength (p \ n)) + 2 * nlength (n + p \ n) +$   
 $2 * nlength (Suc (n + p \ n)) + 7 * nlength (Suc (Suc (2 * n + 2 * p \ n))) +$   
 $(2 * n + 2 * p \ n) * (12 + 2 * nlength (Suc (Suc (2 * n + 2 * p \ n))))$

**shows**  $transforms \ tm12 \ tps0 \ ttt \ tps12$

$\langle proof \rangle$

**abbreviation**  $exc :: nat \Rightarrow config \ where$

$exc \ t \equiv execute \ M \ (start\text{-config} \ 2 \ zs) \ t$

**definition**  $tps13 \equiv tps0$

[9 :=  $exc \ thalt \ <!\> \ 0$ ,  
 11 := ( $\lfloor n \rfloor_N$ , 1),  
 15 := ( $\lfloor p \ n \rfloor_N$ , 1),  
 16 := ( $\lfloor 2 * Suc (n + p \ n) \rfloor_N$ , 1),  
 17 := ( $\lfloor 0 \rfloor_N$ , 1),  
 3 := ( $\lfloor exc \ thalt \ <\#\> \ 0 \rfloor_N$ , 1),  
 4 := ( $\lfloor map (\lambda t. exc \ t \ <\#\> \ 0) [0..<Suc \ thalt] \rfloor_{NL}$ , 1),  
 6 := ( $\lfloor exc \ thalt \ <\#\> \ 1 \rfloor_N$ , 1),  
 7 := ( $\lfloor map (\lambda t. exc \ t \ <\#\> \ 1) [0..<Suc \ thalt] \rfloor_{NL}$ , 1),  
 10 :=  $exc \ thalt \ <!\> \ 1$ ]

**lemma**  $tm13$  [transforms-intros]:

**assumes**  $ttt = 82 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength \ n)^2) +$   
 $3 * nlength (p \ n) + 3 * \max (nlength \ n) (nlength (p \ n)) + 2 * nlength (n + p \ n) +$   
 $2 * nlength (Suc (n + p \ n)) + 7 * nlength (Suc (Suc (2 * n + 2 * p \ n))) +$

$(2 * n + 2 * p * n) * (12 + 2 * nlength (Suc (Suc (2 * n + 2 * p * n)))) +$   
 $(27 + 27 * thalt) * (9 + 2 * nlength thalt)$   
**shows** *transforms tm13 tps0 ttt tps13*  
 ⟨proof⟩

**definition** *tpsA*  $\equiv$  *tps0*

$[9 := exc thalt <!> 0,$   
 $3 := ([exc thalt <\#> 0]_N, 1),$   
 $6 := ([exc thalt <\#> 1]_N, 1),$   
 $10 := exc thalt <!> 1]$

**definition** *tps14*  $\equiv$  *tps0*

$[9 := exc thalt <!> 0,$   
 $11 := ([n]_N, 1),$   
 $15 := ([p n]_N, 1),$   
 $16 := ([2 * Suc (n + p n)]_N, 1),$   
 $17 := ([Suc thalt]_N, 1),$   
 $3 := ([exc thalt <\#> 0]_N, 1),$   
 $4 := ([map (\lambda t. exc t <\#> 0) [0..<Suc thalt]]_{NL}, 1),$   
 $6 := ([exc thalt <\#> 1]_N, 1),$   
 $7 := ([map (\lambda t. exc t <\#> 1) [0..<Suc thalt]]_{NL}, 1),$   
 $10 := exc thalt <!> 1]$

**lemma** *tm14*:

**assumes**  $ttt = 87 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength (p\ n) + 3 * max (nlength\ n) (nlength (p\ n)) + 2 * nlength (n + p\ n) +$   
 $2 * nlength (Suc (n + p\ n)) + 7 * nlength (Suc (Suc (2 * n + 2 * p\ n))) +$   
 $(2 * n + 2 * p * n) * (12 + 2 * nlength (Suc (Suc (2 * n + 2 * p * n)))) +$   
 $(27 + 27 * thalt) * (9 + 2 * nlength thalt) +$   
 $14 * (nlength (map (\lambda t. exc t <\#> 0) [0..<thalt] @ [exc thalt <\#> 0]))^2$   
**shows** *transforms tm14 tps0 ttt tps14*  
 ⟨proof⟩

**definition** *tps14'*  $\equiv$  *tpsA*

$[11 := ([n]_N, 1),$   
 $15 := ([p n]_N, 1),$   
 $16 := ([2 * Suc (n + p n)]_N, 1),$   
 $17 := ([Suc thalt]_N, 1),$   
 $4 := ([map (\lambda t. exc t <\#> 0) [0..<Suc thalt]]_{NL}, 1),$   
 $7 := ([map (\lambda t. exc t <\#> 1) [0..<Suc thalt]]_{NL}, 1)]$

**lemma** *tps14'*: *tps14'* = *tps14*

⟨proof⟩

**lemma** *len-tpsA*: *length tpsA* = *k*

⟨proof⟩

**lemma** *tm14'* [*transforms-intros*]:

**assumes**  $ttt = 87 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength (p\ n) + 3 * max (nlength\ n) (nlength (p\ n)) + 2 * nlength (n + p\ n) +$   
 $2 * nlength (Suc (n + p\ n)) + 7 * nlength (Suc (Suc (2 * n + 2 * p\ n))) +$   
 $(2 * n + 2 * p * n) * (12 + 2 * nlength (Suc (Suc (2 * n + 2 * p * n)))) +$   
 $(27 + 27 * thalt) * (9 + 2 * nlength thalt) +$   
 $14 * (nlength (map (\lambda t. exc t <\#> 0) [0..<thalt] @ [exc thalt <\#> 0]))^2$   
**shows** *transforms tm14 tps0 ttt tps14'*  
 ⟨proof⟩

**definition** *tps15*  $\equiv$  *tpsA*

$[11 := ([n]_N, 1),$   
 $15 := ([p n]_N, 1),$   
 $16 := ([2 * Suc (n + p n)]_N, 1),$   
 $17 := ([thalt]_N, 1),$   
 $4 := ([map (\lambda t. exc t <\#> 0) [0..<Suc thalt]]_{NL}, 1),$

$7 := (\lfloor \text{map } (\lambda t. \text{exc } t <\#\> 1) [0..<\text{Suc } \text{thalt}] \rfloor_{NL}, 1)$

**lemma** *tm15* [*transforms-intros*]:

**assumes**  $\text{ttt} = 95 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * \max\ (nlength\ n)\ (nlength\ (p\ n)) + 2 * nlength\ (n + p\ n) +$   
 $2 * nlength\ (\text{Suc}\ (n + p\ n)) + 7 * nlength\ (\text{Suc}\ (\text{Suc}\ (2 * n + 2 * p\ n))) +$   
 $(2 * n + 2 * p\ n) * (12 + 2 * nlength\ (\text{Suc}\ (\text{Suc}\ (2 * n + 2 * p\ n)))) +$   
 $(27 + 27 * \text{thalt}) * (9 + 2 * nlength\ \text{thalt}) +$   
 $14 * (nlength\ (\text{map}\ (\lambda t. \text{exc } t <\#\> 0) [0..<\text{thalt}] @ [\text{exc } \text{thalt } <\#\> 0]))^2 +$   
 $2 * nlength\ (\text{Suc } \text{thalt})$

**shows** *transforms tm15 tps0 ttt tps15*

*<proof>*

**definition** *tps16*  $\equiv$  *tpsA*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p\ n \rfloor_N, 1),$   
 $16 := (\lfloor 2 * \text{Suc}\ (n + p\ n) \rfloor_N, 1),$   
 $17 := (\lfloor \text{thalt} \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } t <\#\> 0) [0..<\text{Suc } \text{thalt}] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } t <\#\> 1) [0..<\text{Suc } \text{thalt}] \rfloor_{NL}, 1),$   
 $18 := (\lfloor 2 * \text{Suc}\ (n + p\ n) \rfloor_N, 1)]$

**lemma** *tm16* [*transforms-intros*]:

**assumes**  $\text{ttt} = 109 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * \max\ (nlength\ n)\ (nlength\ (p\ n)) + 2 * nlength\ (n + p\ n) +$   
 $2 * nlength\ (\text{Suc}\ (n + p\ n)) + 10 * nlength\ (\text{Suc}\ (\text{Suc}\ (2 * n + 2 * p\ n))) +$   
 $(2 * n + 2 * p\ n) * (12 + 2 * nlength\ (\text{Suc}\ (\text{Suc}\ (2 * n + 2 * p\ n)))) +$   
 $(27 + 27 * \text{thalt}) * (9 + 2 * nlength\ \text{thalt}) +$   
 $14 * (nlength\ (\text{map}\ (\lambda t. \text{exc } t <\#\> 0) [0..<\text{thalt}] @ [\text{exc } \text{thalt } <\#\> 0]))^2 +$   
 $2 * nlength\ (\text{Suc } \text{thalt})$

**shows** *transforms tm16 tps0 ttt tps16*

*<proof>*

**definition** *tps17*  $\equiv$  *tpsA*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p\ n \rfloor_N, 1),$   
 $16 := (\lfloor 2 * \text{Suc}\ (n + p\ n) \rfloor_N, 1),$   
 $17 := (\lfloor \text{thalt} \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } t <\#\> 0) [0..<\text{Suc } \text{thalt}] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } t <\#\> 1) [0..<\text{Suc } \text{thalt}] \rfloor_{NL}, 1),$   
 $18 := (\lfloor \text{Suc}\ (2 * \text{Suc}\ (n + p\ n)) \rfloor_N, 1)]$

**lemma** *tm17* [*transforms-intros*]:

**assumes**  $\text{ttt} = 114 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * \max\ (nlength\ n)\ (nlength\ (p\ n)) + 2 * nlength\ (n + p\ n) +$   
 $2 * nlength\ (\text{Suc}\ (n + p\ n)) + 10 * nlength\ (\text{Suc}\ (\text{Suc}\ (2 * n + 2 * p\ n))) +$   
 $(2 * n + 2 * p\ n) * (12 + 2 * nlength\ (\text{Suc}\ (\text{Suc}\ (2 * n + 2 * p\ n)))) +$   
 $(27 + 27 * \text{thalt}) * (9 + 2 * nlength\ \text{thalt}) +$   
 $14 * (nlength\ (\text{map}\ (\lambda t. \text{exc } t <\#\> 0) [0..<\text{thalt}] @ [\text{exc } \text{thalt } <\#\> 0]))^2 +$   
 $2 * nlength\ (\text{Suc } \text{thalt}) + 2 * nlength\ (\text{Suc}\ (\text{Suc}\ (2 * n + 2 * p\ n)))$

**shows** *transforms tm17 tps0 ttt tps17*

*<proof>*

**definition** *tps18*  $\equiv$  *tpsA*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p\ n \rfloor_N, 1),$   
 $16 := (\lfloor 2 * \text{Suc}\ (n + p\ n) \rfloor_N, 1),$   
 $17 := (\lfloor \text{thalt} \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } t <\#\> 0) [0..<\text{Suc } \text{thalt}] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } t <\#\> 1) [0..<\text{Suc } \text{thalt}] \rfloor_{NL}, 1),$   
 $18 := (\lfloor \text{thalt} + \text{Suc}\ (2 * \text{Suc}\ (n + p\ n)) \rfloor_N, 1)]$

**lemma** *tm18* [*transforms-intros*]:

**assumes**  $t_{tt} = 124 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * max\ (nlength\ n)\ (nlength\ (p\ n)) + 2 * nlength\ (n + p\ n) +$   
 $2 * nlength\ (Suc\ (n + p\ n)) + 10 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n))) +$   
 $(2 * n + 2 * p\ n) * (12 + 2 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n)))) +$   
 $(27 + 27 * thalt) * (9 + 2 * nlength\ thalt) +$   
 $14 * (nlength\ (map\ (\lambda t. exc\ t <\#\> 0)\ [0..<thalt]\ @\ [exc\ thalt <\#\> 0]))^2 +$   
 $2 * nlength\ (Suc\ thalt) + 2 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n))) +$   
 $3 * max\ (nlength\ thalt)\ (nlength\ (Suc\ (Suc\ (Suc\ (2 * n + 2 * p\ n))))))$   
**shows** *transforms tm18 tps0 ttt tps18*  
 ⟨proof⟩

**definition**  $tps19 \equiv tpsA$

[11 := ( $\lfloor n \rfloor_N, 1$ ),  
 15 := ( $\lfloor p\ n \rfloor_N, 1$ ),  
 16 := ( $\lfloor 2 * Suc\ (n + p\ n) \rfloor_N, 1$ ),  
 17 := ( $\lfloor thalt \rfloor_N, 1$ ),  
 4 := ( $\lfloor map\ (\lambda t. exc\ t <\#\> 0)\ [0..<Suc\ thalt]\ \rfloor_{NL}, 1$ ),  
 7 := ( $\lfloor map\ (\lambda t. exc\ t <\#\> 1)\ [0..<Suc\ thalt]\ \rfloor_{NL}, 1$ ),  
 18 := ( $\lfloor thalt + Suc\ (2 * Suc\ (n + p\ n)) \rfloor_N, 1$ ),  
 19 := ( $\lfloor max\ G\ (length\ M) \rfloor_N, 1$ )]

**lemma**  $tm19$  [*transforms-intros*]:

**assumes**  $t_{tt} = 134 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * max\ (nlength\ n)\ (nlength\ (p\ n)) + 2 * nlength\ (n + p\ n) +$   
 $2 * nlength\ (Suc\ (n + p\ n)) + 10 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n))) +$   
 $(2 * n + 2 * p\ n) * (12 + 2 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n)))) +$   
 $(27 + 27 * thalt) * (9 + 2 * nlength\ thalt) +$   
 $14 * (nlength\ (map\ (\lambda t. exc\ t <\#\> 0)\ [0..<thalt]\ @\ [exc\ thalt <\#\> 0]))^2 +$   
 $2 * nlength\ (Suc\ thalt) + 2 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n))) +$   
 $3 * max\ (nlength\ thalt)\ (nlength\ (Suc\ (Suc\ (Suc\ (2 * n + 2 * p\ n)))))) +$   
 $2 * nlength\ (max\ G\ (length\ M))$   
**shows** *transforms tm19 tps0 ttt tps19*  
 ⟨proof⟩

**definition**  $tps20 \equiv tpsA$

[11 := ( $\lfloor n \rfloor_N, 1$ ),  
 15 := ( $\lfloor p\ n \rfloor_N, 1$ ),  
 16 := ( $\lfloor 2 * Suc\ (n + p\ n) \rfloor_N, 1$ ),  
 17 := ( $\lfloor thalt \rfloor_N, 1$ ),  
 4 := ( $\lfloor map\ (\lambda t. exc\ t <\#\> 0)\ [0..<Suc\ thalt]\ \rfloor_{NL}, 1$ ),  
 7 := ( $\lfloor map\ (\lambda t. exc\ t <\#\> 1)\ [0..<Suc\ thalt]\ \rfloor_{NL}, 1$ ),  
 18 := ( $\lfloor thalt + Suc\ (2 * Suc\ (n + p\ n)) \rfloor_N, 1$ ),  
 19 := ( $\lfloor max\ G\ (length\ M) \rfloor_N, 1$ ),  
 20 := ( $\lfloor (thalt + Suc\ (2 * Suc\ (n + p\ n))) * max\ G\ (length\ M) \rfloor_N, 1$ )]

**lemma**  $tm20$ :

**assumes**  $t_{tt} = 138 + 11 * n^2 + (d\text{-polynomial } p + d\text{-polynomial } p * (nlength\ n)^2) +$   
 $3 * nlength\ (p\ n) + 3 * max\ (nlength\ n)\ (nlength\ (p\ n)) + 2 * nlength\ (n + p\ n) +$   
 $2 * nlength\ (Suc\ (n + p\ n)) + 10 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n))) +$   
 $(2 * n + 2 * p\ n) * (12 + 2 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n)))) +$   
 $(27 + 27 * thalt) * (9 + 2 * nlength\ thalt) +$   
 $14 * (nlength\ (map\ (\lambda t. exc\ t <\#\> 0)\ [0..<thalt]\ @\ [exc\ thalt <\#\> 0]))^2 +$   
 $2 * nlength\ (Suc\ thalt) + 2 * nlength\ (Suc\ (Suc\ (2 * n + 2 * p\ n))) +$   
 $3 * max\ (nlength\ thalt)\ (nlength\ (Suc\ (Suc\ (Suc\ (2 * n + 2 * p\ n)))))) +$   
 $2 * nlength\ (max\ G\ (length\ M)) +$   
 $26 * (nlength\ (Suc\ (Suc\ (Suc\ (thalt + (2 * n + 2 * p\ n)))))) + nlength\ (max\ G\ (length\ M)) *$   
 $(nlength\ (Suc\ (Suc\ (Suc\ (thalt + (2 * n + 2 * p\ n)))))) + nlength\ (max\ G\ (length\ M))$   
**shows** *transforms tm20 tps0 ttt tps20*  
 ⟨proof⟩

**lemma**  $tm20'$  [*transforms-intros*]:

**assumes**  $t_{tt} = (2 * d\text{-polynomial } p + 826) * (max\ G\ (length\ M) + thalt + Suc\ (Suc\ (Suc\ (2 * n + 2 * p\ n)))) \wedge 4$

**shows** *transforms tm20 tps0 ttt tps20*  
 (proof)

**end**

**end**

**lemma** *transforms-tm-PHI-initI*:

**fixes**  $G :: \text{nat}$  **and**  $M :: \text{machine}$  **and**  $p :: \text{nat} \Rightarrow \text{nat}$

**fixes**  $k$   $H$   $\text{thalt} :: \text{nat}$  **and**  $\text{tps}$   $\text{tps}' :: \text{tape list}$  **and**  $xs$   $zs :: \text{symbol list}$

**assumes** *poly-p: polynomial p*

**and**  $M\text{-tm}$ : *turing-machine 2 G M*

**and**  $k$ :  $k = \text{length tps } 20 < k$

**and**  $H$ :  $H = \text{max } G (\text{length } M)$

**and**  $xs$ : *bit-symbols xs*

**and**  $zs$ :  $zs = 2 \# 2 \# \text{replicate } (2 * \text{length } xs + 2 * p (\text{length } xs)) \ 2$

**assumes** *thalt*:

$\forall t < \text{thalt}. \text{fst } (\text{execute } M (\text{start-config } 2 \text{ } zs) \ t) < \text{length } M$

$\text{fst } (\text{execute } M (\text{start-config } 2 \text{ } zs) \ \text{thalt}) = \text{length } M$

**assumes** *tps0*:

$\text{tps} ! 0 = (\lfloor xs \rfloor, 1)$

$\bigwedge i. 0 < i \Rightarrow i \leq 10 \Rightarrow \text{tps} ! i = (\lfloor \square \rfloor, 0)$

$\bigwedge i. 10 < i \Rightarrow i < k \Rightarrow \text{tps} ! i = (\lfloor \square \rfloor, 1)$

**assumes**  $\text{ttt} = (2 * d\text{-polynomial } p + 826) * (\text{max } G (\text{length } M) + \text{thalt} + \text{Suc } (\text{Suc } (\text{Suc } (2 * (\text{length } xs) + 2 * p (\text{length } xs)))))) \wedge 4$

**assumes**  $\text{tps}' = \text{tps}$

$9 := \text{execute } M (\text{start-config } 2 \text{ } zs) \ \text{thalt} \ <!\> \ 0,$

$3 := (\lfloor \text{execute } M (\text{start-config } 2 \text{ } zs) \ \text{thalt} \ <\#\> \ 0 \rfloor_N, 1),$

$6 := (\lfloor \text{execute } M (\text{start-config } 2 \text{ } zs) \ \text{thalt} \ <\#\> \ 1 \rfloor_N, 1),$

$10 := \text{execute } M (\text{start-config } 2 \text{ } zs) \ \text{thalt} \ <!\> \ 1,$

$11 := (\lfloor \text{length } xs \rfloor_N, 1),$

$15 := (\lfloor p (\text{length } xs) \rfloor_N, 1),$

$16 := (\lfloor 2 * \text{Suc } ((\text{length } xs) + p (\text{length } xs)) \rfloor_N, 1),$

$17 := (\lfloor \text{thalt} \rfloor_N, 1),$

$4 := (\lfloor \text{map } (\lambda t. \text{execute } M (\text{start-config } 2 \text{ } zs) \ t \ <\#\> \ 0) \ [0..<\text{Suc } \text{thalt}]\rfloor_{NL}, 1),$

$7 := (\lfloor \text{map } (\lambda t. \text{execute } M (\text{start-config } 2 \text{ } zs) \ t \ <\#\> \ 1) \ [0..<\text{Suc } \text{thalt}]\rfloor_{NL}, 1),$

$18 := (\lfloor \text{thalt} + \text{Suc } (2 * \text{Suc } ((\text{length } xs) + p (\text{length } xs))) \rfloor_N, 1),$

$19 := (\lfloor \text{max } G (\text{length } M) \rfloor_N, 1),$

$20 := (\lfloor (\text{thalt} + \text{Suc } (2 * \text{Suc } ((\text{length } xs) + p (\text{length } xs)))) * \text{max } G (\text{length } M) \rfloor_N, 1)$

**shows** *transforms (tm-PHI-init G M p) tps ttt tps'*

(proof)

Next we transfer the semantics of *tm-PHI-init* to the locale *reduction-sat-x*.

**lemma** (*in reduction-sat-x*) *tm-PHI-init* [*transforms-intros*]:

**fixes**  $k :: \text{nat}$  **and**  $\text{tps}$   $\text{tps}' :: \text{tape list}$

**assumes**  $k = \text{length tps}$  **and**  $20 < k$

**assumes**

$\text{tps} ! 0 = (\lfloor xs \rfloor, 1)$

$\bigwedge i. 0 < i \Rightarrow i \leq 10 \Rightarrow \text{tps} ! i = (\lfloor \square \rfloor, 0)$

$\bigwedge i. 10 < i \Rightarrow i < k \Rightarrow \text{tps} ! i = (\lfloor \square \rfloor, 1)$

**assumes**  $\text{ttt} = (2 * d\text{-polynomial } p + 826) * (H + T' + \text{Suc } (\text{Suc } (\text{Suc } (2 * n + 2 * p \ n)))) \wedge 4$

**assumes**  $\text{tps}' = \text{tps}$

$9 := \text{exc } (\text{zeros } m) \ T' \ <!\> \ 0,$

$3 := (\lfloor \text{exc } (\text{zeros } m) \ T' \ <\#\> \ 0 \rfloor_N, 1),$

$6 := (\lfloor \text{exc } (\text{zeros } m) \ T' \ <\#\> \ 1 \rfloor_N, 1),$

$10 := \text{exc } (\text{zeros } m) \ T' \ <!\> \ 1,$

$11 := (\lfloor n \rfloor_N, 1),$

$15 := (\lfloor p \ n \rfloor_N, 1),$

$16 := (\lfloor 2 * \text{Suc } (n + p \ n) \rfloor_N, 1),$

$17 := (\lfloor T' \rfloor_N, 1),$

$4 := (\lfloor \text{map } (\lambda t. \text{exc } (\text{zeros } m) \ t \ <\#\> \ 0) \ [0..<\text{Suc } T']\rfloor_{NL}, 1),$

$7 := (\lfloor \text{map } (\lambda t. \text{exc } (\text{zeros } m) \ t \ <\#\> \ 1) \ [0..<\text{Suc } T']\rfloor_{NL}, 1),$

$18 := (\lfloor T' + \text{Suc } (2 * \text{Suc } (n + p \ n)) \rfloor_N, 1),$

$19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor (T' + \text{Suc } (2 * \text{Suc } (n + p n))) * H \rfloor_N, 1)$   
**shows** *transforms* (*tm-PHI-init* *G M p*) *tps ttt tps'*  
*<proof>*

### 8.3 The actual Turing machine computing the reduction

In this section we put everything together to build a Turing machine that given a string  $x$  outputs the CNF formula  $\Phi$  defined in Chapter 6. In principle this is just a sequence of the TMs *tm-PHI-init*, *tm-PHI0*,  $\dots$ , *tm-PHI9*, where *tm-PHI345* occurs once for each of the formulas  $\Phi_3$ ,  $\Phi_4$ , and  $\Phi_5$ . All these TMs are linked by TMs that copy values prepared by *tm-PHI-init* to the tapes where the following TM expects them. Also as the very first step the tape heads on tapes 0 and 11 and beyond must be moved one cell to the right to meet *tm-PHI-init*'s expectations.

The TM will have 110 tapes because we just allocate another batch of tapes for every TM computing a  $\Phi_i$ , rather than cleaning up and reusing tapes.

The Turing machine for computing  $\Phi$  is to be defined in the locale *reduction-sat*. We save the space to write the TM in closed form.

**context** *reduction-sat*  
**begin**

**definition**  $tm1 \equiv tm\text{-right-many } \{i. i < 1 \vee 10 < i\}$   
**definition**  $tm2 \equiv tm1 ;; tm\text{-PHI-init } G M p$

**definition**  $tm3 \equiv tm2 ;; tm\text{-copyn } 18\ 21$   
**definition**  $tm4 \equiv tm3 ;; tm\text{-copyn } 19\ 22$   
**definition**  $tm5 \equiv tm4 ;; tm\text{-right } 1$   
**definition**  $tm6 \equiv tm5 ;; tm\text{-PHI0 } 21$

**definition**  $tm7 \equiv tm6 ;; tm\text{-setn } 29\ H$   
**definition**  $tm8 \equiv tm7 ;; tm\text{-PHI1 } 28$

**definition**  $tm9 \equiv tm8 ;; tm\text{-copyn } 11\ 35$   
**definition**  $tm10 \equiv tm9 ;; tm\text{-setn } 36\ H$   
**definition**  $tm11 \equiv tm10 ;; tm\text{-PHI2 } 35$

**definition**  $tm12 \equiv tm11 ;; tm\text{-setn } 42\ 1$   
**definition**  $tm13 \equiv tm12 ;; tm\text{-setn } 43\ H$   
**definition**  $tm14 \equiv tm13 ;; tm\text{-setn } 44\ 2$   
**definition**  $tm15 \equiv tm14 ;; tm\text{-copyn } 11\ 50$   
**definition**  $tm16 \equiv tm15 ;; tm\text{-times2incr } 50$   
**definition**  $tm17 \equiv tm16 ;; tm\text{-PHI345 } 2\ 42$

**definition**  $tm18 \equiv tm17 ;; tm\text{-setn } 52\ H$   
**definition**  $tm19 \equiv tm18 ;; tm\text{-setn } 53\ 2$   
**definition**  $tm20 \equiv tm19 ;; tm\text{-copyn } 11\ 51$   
**definition**  $tm21 \equiv tm20 ;; tm\text{-times2 } 51$   
**definition**  $tm22 \equiv tm21 ;; tm\text{-plus-const } 3\ 51$   
**definition**  $tm23 \equiv tm22 ;; tm\text{-copyn } 16\ 59$   
**definition**  $tm24 \equiv tm23 ;; tm\text{-incr } 59$   
**definition**  $tm25 \equiv tm24 ;; tm\text{-PHI345 } 2\ 51$

**definition**  $tm26 \equiv tm25 ;; tm\text{-setn } 61\ H$   
**definition**  $tm27 \equiv tm26 ;; tm\text{-copyn } 16\ 60$   
**definition**  $tm28 \equiv tm27 ;; tm\text{-incr } 60$   
**definition**  $tm29 \equiv tm28 ;; tm\text{-copyn } 60\ 68$   
**definition**  $tm30 \equiv tm29 ;; tm\text{-add } 17\ 68$   
**definition**  $tm31 \equiv tm30 ;; tm\text{-PHI345 } 1\ 60$

**definition**  $tm32 \equiv tm31 ;; tm\text{-setn } 69\ 2$   
**definition**  $tm33 \equiv tm32 ;; tm\text{-setn } 70\ H$   
**definition**  $tm34 \equiv tm33 ;; tm\text{-PHI6 } 69$

```

definition tm35 ≡ tm34 ;; tm-copyn 11 77
definition tm36 ≡ tm35 ;; tm-times2 77
definition tm37 ≡ tm36 ;; tm-plus-const 4 77
definition tm38 ≡ tm37 ;; tm-setn 78 H
definition tm39 ≡ tm38 ;; tm-copyn 15 83
definition tm40 ≡ tm39 ;; tm-PHI7 77

definition tm41 ≡ tm40 ;; tm-copyn 18 84
definition tm42 ≡ tm41 ;; tm-add 17 84
definition tm43 ≡ tm42 ;; tm-add 17 84
definition tm44 ≡ tm43 ;; tm-add 17 84
definition tm45 ≡ tm44 ;; tm-incr 84
definition tm46 ≡ tm45 ;; tm-setn 85 H
definition tm47 ≡ tm46 ;; tm-PHI8 84

definition tm48 ≡ tm47 ;; tm-copyn 20 91
definition tm49 ≡ tm48 ;; tm-setn 92 H
definition tm50 ≡ tm49 ;; tm-setn 93 Z
definition tm51 ≡ tm50 ;; tm-copyn 17 94
definition tm52 ≡ tm51 ;; tm-set 95 (numlistlist (formula-n ψ))
definition tm53 ≡ tm52 ;; tm-set 96 (numlistlist (formula-n ψ'))
definition tm54 ≡ tm53 ;; tm-setn 97 1
definition tm55 ≡ tm54 ;; tm-PHI9 4 7 91

definition tm56 ≡ tm55 ;; tm-cr 1
definition tm57 ≡ tm56 ;; tm-cp-until 1 109 {0}
definition tm58 ≡ tm57 ;; tm-erase-cr 1
definition tm59 ≡ tm58 ;; tm-cr 109
definition tm60 ≡ tm59 ;; tm-binencode 109 1

definition H' :: nat where
  H' ≡ Suc (Suc H)

lemma H-gr-3: H > 3
  ⟨proof⟩

lemma H': H' ≥ Suc (length M) H' ≥ G H' ≥ 6
  ⟨proof⟩

lemma tm40-tm: turing-machine 110 H' tm40
  ⟨proof⟩

lemma tm55-tm: turing-machine 110 H' tm55
  ⟨proof⟩

lemma tm60-tm: turing-machine 110 H' tm60
  ⟨proof⟩

end

Unlike before, we prove the semantics inside locale reduction-sat-x since we need not be concerned with
“polluting” the namespace of the locale. After all there will not be any more Turing machines.

context reduction-sat-x
begin

context
  fixes tps0 :: tape list
  assumes k: 110 = length tps0
  assumes tps0:
    tps0 ! 0 = (|xs|, 0)
    ⋀i. 0 < i ⇒ i < 110 ⇒ tps0 ! i = (|[]|, 0)
begin

```

**definition**  $tps1 \equiv \text{map } (\lambda j. \text{if } j < 1 \vee 10 < j \text{ then } tps0 ! j |+| 1 \text{ else } tps0 ! j) [0..<110]$

**lemma**  $\text{lentps1}: \text{length } tps1 = 110$

$\langle \text{proof} \rangle$

**lemma**  $tps1$ :

$0 < j \implies j < 10 \implies tps1 ! j = ([\square], 0)$

$10 < j \implies j < 110 \implies tps1 ! j = ([\square], 1)$

$\langle \text{proof} \rangle$

**lemma**  $tps1'$ :  $tps1 ! 0 = (\lfloor xs \rfloor, 1)$

$\langle \text{proof} \rangle$

**lemma**  $tm1$  [*transforms-intros*]: *transforms*  $tm1$   $tps0$  1  $tps1$

$\langle \text{proof} \rangle$

**abbreviation**  $zs \equiv \text{zeros } m$

**definition**  $tpsA \equiv tps1$

$[9 := \text{exc } zs \ T' <!\> 0,$

$3 := (\lfloor \text{exc } zs \ T' <\#\> 0 \rfloor_N, 1),$

$6 := (\lfloor \text{exc } zs \ T' <\#\> 1 \rfloor_N, 1),$

$10 := \text{exc } zs \ T' <!\> 1]$

**lemma**  $tpsA$ :

$tpsA ! 0 = (\lfloor xs \rfloor, 1)$

$tpsA ! 1 = ([\square], 0)$

$10 < j \implies j < 110 \implies tpsA ! j = ([\square], 1)$

$\langle \text{proof} \rangle$

**lemma**  $\text{lentpsA}: \text{length } tpsA = 110$

$\langle \text{proof} \rangle$

**definition**  $tps2 \equiv tpsA$

$[11 := (\lfloor n \rfloor_N, 1),$

$15 := (\lfloor p \ n \rfloor_N, 1),$

$16 := (\lfloor m \rfloor_N, 1),$

$17 := (\lfloor T' \rfloor_N, 1),$

$4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t <\#\> 0) [0..<Suc \ T'] \rfloor_{NL}, 1),$

$7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t <\#\> 1) [0..<Suc \ T'] \rfloor_{NL}, 1),$

$18 := (\lfloor m' \rfloor_N, 1),$

$19 := (\lfloor H \rfloor_N, 1),$

$20 := (\lfloor m' * H \rfloor_N, 1)]$

**lemma**  $\text{lentps2}: \text{length } tps2 = 110$

$\langle \text{proof} \rangle$

**lemma**  $tm2$  [*transforms-intros*]:

**assumes**  $ttt = 1 + (2 * d\text{-polynomial } p + 826) * (H + m') \wedge 4$

**shows** *transforms*  $tm2$   $tps0$   $ttt$   $tps2$

$\langle \text{proof} \rangle$

**definition**  $tps3 \equiv tpsA$

$[11 := (\lfloor n \rfloor_N, 1),$

$15 := (\lfloor p \ n \rfloor_N, 1),$

$16 := (\lfloor m \rfloor_N, 1),$

$17 := (\lfloor T' \rfloor_N, 1),$

$4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t <\#\> 0) [0..<Suc \ T'] \rfloor_{NL}, 1),$

$7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t <\#\> 1) [0..<Suc \ T'] \rfloor_{NL}, 1),$

$18 := (\lfloor m' \rfloor_N, 1),$

$19 := (\lfloor H \rfloor_N, 1),$

$20 := (\lfloor m' * H \rfloor_N, 1),$

21 := ( $\lfloor m' \rfloor_N, 1$ )

**lemma** *lentps3*: *length tps3 = 110*  
(*proof*)

**lemma** *tm3* [*transforms-intros*]:

**assumes**  $ttt = 15 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * nlength\ m'$

**shows** *transforms tm3 tps0 ttt tps3*

(*proof*)

**definition** *tps4*  $\equiv$  *tpsA*

[11 := ( $\lfloor n \rfloor_N, 1$ ),

15 := ( $\lfloor p\ n \rfloor_N, 1$ ),

16 := ( $\lfloor m \rfloor_N, 1$ ),

17 := ( $\lfloor T' \rfloor_N, 1$ ),

4 := ( $\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T'] \rfloor_{NL}, 1$ ),

7 := ( $\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T'] \rfloor_{NL}, 1$ ),

18 := ( $\lfloor m' \rfloor_N, 1$ ),

19 := ( $\lfloor H \rfloor_N, 1$ ),

20 := ( $\lfloor m' * H \rfloor_N, 1$ ),

21 := ( $\lfloor m' \rfloor_N, 1$ ),

22 := ( $\lfloor H \rfloor_N, 1$ )

**lemma** *lentps4*: *length tps4 = 110*  
(*proof*)

**lemma** *tm4* [*transforms-intros*]:

**assumes**  $ttt = 29 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * nlength\ m' + 3 * nlength\ H$

**shows** *transforms tm4 tps0 ttt tps4*

(*proof*)

**definition** *tps5*  $\equiv$  *tpsA*

[11 := ( $\lfloor n \rfloor_N, 1$ ),

15 := ( $\lfloor p\ n \rfloor_N, 1$ ),

16 := ( $\lfloor m \rfloor_N, 1$ ),

17 := ( $\lfloor T' \rfloor_N, 1$ ),

4 := ( $\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T'] \rfloor_{NL}, 1$ ),

7 := ( $\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T'] \rfloor_{NL}, 1$ ),

18 := ( $\lfloor m' \rfloor_N, 1$ ),

19 := ( $\lfloor H \rfloor_N, 1$ ),

20 := ( $\lfloor m' * H \rfloor_N, 1$ ),

21 := ( $\lfloor m' \rfloor_N, 1$ ),

22 := ( $\lfloor H \rfloor_N, 1$ ),

1 := ( $\lfloor [] \rfloor, 1$ )

**lemma** *lentps5*: *length tps5 = 110*  
(*proof*)

**lemma** *tm5* [*transforms-intros*]:

**assumes**  $ttt = 30 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * nlength\ m' + 3 * nlength\ H$

**shows** *transforms tm5 tps0 ttt tps5*

(*proof*)

**definition** *tps6*  $\equiv$  *tpsA*

[11 := ( $\lfloor n \rfloor_N, 1$ ),

15 := ( $\lfloor p\ n \rfloor_N, 1$ ),

16 := ( $\lfloor m \rfloor_N, 1$ ),

17 := ( $\lfloor T' \rfloor_N, 1$ ),

4 := ( $\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T'] \rfloor_{NL}, 1$ ),

7 := ( $\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T'] \rfloor_{NL}, 1$ ),

18 := ( $\lfloor m' \rfloor_N, 1$ ),

19 := ( $\lfloor H \rfloor_N, 1$ ),

20 := ( $\lfloor m' * H \rfloor_N, 1$ ),

$21 := (\lfloor m' \rfloor_N, 1),$   
 $22 := (\lfloor H \rfloor_N, 1),$   
 $1 := \text{niltape}(\text{formula-}n \Phi_0),$   
 $21 := (\lfloor \text{Suc}(\text{Suc } m') \rfloor_N, 1),$   
 $21 + 2 := (\lfloor 0 \rfloor_N, 1),$   
 $21 + 6 := (\lfloor \text{nil-Psi}(\text{Suc}(\text{Suc } m') * H) H 0 \rfloor_{NLL}, 1)]$

**lemma** *lentps6*:  $\text{length } \textit{tps6} = 110$   
*<proof>*

**lemma** *tm6* [*transforms-intros*]:

**assumes**  $\textit{ttt} = 30 + (2 * d\text{-polynomial } p + 826) * (H + m')^{\wedge} 4 + 3 * \text{nlength } m' + 3 * \text{nlength } H +$   
 $5627 * H^{\wedge} 4 * (3 + \text{nlength}(3 * H + m' * H))^2$   
**shows** *transforms tm6 tps0 ttt tps6*  
*<proof>*

**definition** *tpsB*  $\equiv \textit{tpsA}$

$[21 := (\lfloor m' \rfloor_N, 1),$   
 $22 := (\lfloor H \rfloor_N, 1),$   
 $21 := (\lfloor \text{Suc}(\text{Suc } m') \rfloor_N, 1),$   
 $21 + 2 := (\lfloor 0 \rfloor_N, 1),$   
 $21 + 6 := (\lfloor \text{nil-Psi}(\text{Suc}(\text{Suc } m') * H) H 0 \rfloor_{NLL}, 1)]$

**lemma** *tpsB*:  $j > 27 \implies j < 110 \implies \textit{tpsB} ! j = (\lfloor \square \rfloor, 1)$   
*<proof>*

**lemma** *lentpsB*:  $\text{length } \textit{tpsB} = 110$   
*<proof>*

**lemma** *tps6*:  $\textit{tps6} = \textit{tpsB}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map}(\lambda t. \text{exc } z s t <\#\#> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map}(\lambda t. \text{exc } z s t <\#\#> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{niltape}(\text{formula-}n \Phi_0)]$   
*<proof>*

**lemma** *tps6'*:  $j > 27 \implies j < 110 \implies \textit{tps6}' ! j = (\lfloor \square \rfloor, 1)$   
*<proof>*

**definition** *tps7*  $\equiv \textit{tpsB}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map}(\lambda t. \text{exc } z s t <\#\#> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map}(\lambda t. \text{exc } z s t <\#\#> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{niltape}(\text{formula-}n \Phi_0),$   
 $29 := (\lfloor H \rfloor_N, 1)]$

**lemma** *tm7* [*transforms-intros*]:

**assumes**  $\textit{ttt} = 40 + (2 * d\text{-polynomial } p + 826) * (H + m')^{\wedge} 4 + 3 * \text{nlength } m' + 3 * \text{nlength } H +$   
 $5627 * H^{\wedge} 4 * (3 + \text{nlength}(3 * H + m' * H))^2 + 2 * \text{nlength } H$   
**shows** *transforms tm7 tps0 ttt tps7*  
*<proof>*



$3 * \text{nlength } n$   
**shows** *transforms tm9 tps0 ttt tps9*  
 ⟨*proof*⟩

**definition** *tps10*  $\equiv$  *tpsC*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } (\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1),$   
 $35 := (\lfloor n \rfloor_N, 1),$   
 $36 := (\lfloor H \rfloor_N, 1)]$

**lemma** *tm10* [*transforms-intros*]:

**assumes**  $\text{ttt} = 64 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 +$   
 $3 * \text{nlength } m' + 5 * \text{nlength } H + 5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 +$   
 $1875 * H^4 + 3 * \text{nlength } n + 2 * \text{nlength } H$   
**shows** *transforms tm10 tps0 ttt tps10*  
 ⟨*proof*⟩

**definition** *tps11*  $\equiv$  *tpsC*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } ((\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1) \ @ \ \text{formula-}n \ \Phi_2),$   
 $35 := (\lfloor n \rfloor_N, 1),$   
 $36 := (\lfloor H \rfloor_N, 1),$   
 $35 := (\lfloor 2 * n + 2 \rfloor_N, 1),$   
 $35 + 2 := (\lfloor 3 \rfloor_N, 1),$   
 $35 + 6 := (\lfloor \text{nll-Psi } (\text{Suc } (\text{Suc } (2 * n)) * H) \ H \ 3 \rfloor_{NLL}, 1)]$

**lemma** *tm11* [*transforms-intros*]:

**assumes**  $\text{ttt} = 64 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 +$   
 $3 * \text{nlength } m' + 7 * \text{nlength } H + 5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 +$   
 $1875 * H^4 + 3 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2$   
**shows** *transforms tm11 tps0 ttt tps11*  
 ⟨*proof*⟩

**definition** *tpsD*  $\equiv$  *tpsC*

$[35 := (\lfloor n \rfloor_N, 1),$   
 $36 := (\lfloor H \rfloor_N, 1),$   
 $35 := (\lfloor 2 * n + 2 \rfloor_N, 1),$   
 $35 + 2 := (\lfloor 3 \rfloor_N, 1),$   
 $35 + 6 := (\lfloor \text{nll-Psi } (\text{Suc } (\text{Suc } (2 * n)) * H) \ H \ 3 \rfloor_{NLL}, 1)]$

**lemma** *tpsD*:  $41 < j \implies j < 110 \implies \text{tpsD} ! j = (\lfloor \square \rfloor, 1)$   
 ⟨*proof*⟩

**lemma** *lentpsD*:  $\text{length } \text{tpsD} = 110$   
 ⟨*proof*⟩

**lemma** *tps11*:  $\text{tps11} = \text{tpsD}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } ((\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1) \ @ \ \text{formula-}n \ \Phi_2)$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tps12} \equiv \text{tpsD}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } ((\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1) \ @ \ \text{formula-}n \ \Phi_2),$   
 $42 := (\lfloor 1 \rfloor_N, 1)$

**lemma**  $\text{tm12}$  [*transforms-intros*]:

**assumes**  $\text{tnt} = 76 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 7 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $3 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2$   
**shows** *transforms tm12 tps0 tnt tps12*  
 $\langle \text{proof} \rangle$

**definition**  $\text{tps13} \equiv \text{tpsD}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } ((\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1) \ @ \ \text{formula-}n \ \Phi_2),$   
 $42 := (\lfloor 1 \rfloor_N, 1),$   
 $43 := (\lfloor H \rfloor_N, 1)$

**lemma**  $\text{tm13}$  [*transforms-intros*]:

**assumes**  $\text{tnt} = 86 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 9 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $3 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2$   
**shows** *transforms tm13 tps0 tnt tps13*  
 $\langle \text{proof} \rangle$

**definition**  $\text{tps14} \equiv \text{tpsD}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$

$1 := \text{nlltape} ((\text{formula-}n \Phi_0 @ \text{formula-}n \Phi_1) @ \text{formula-}n \Phi_2),$   
 $42 := ([1]_N, 1),$   
 $43 := ([H]_N, 1),$   
 $44 := ([2]_N, 1)]$

**lemma** *tm14* [*transforms-intros*]:

**assumes**  $ttt = 100 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 9 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $3 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2$   
**shows** *transforms tm14 tps0 ttt tps14*  
 ⟨*proof*⟩

**definition** *tps15*  $\equiv$  *tpsD*

$[11 := ([n]_N, 1),$   
 $15 := ([p \ n]_N, 1),$   
 $16 := ([m]_N, 1),$   
 $17 := ([T']_N, 1),$   
 $4 := ([\text{map } (\lambda t. \text{exc } zs \ t <\#\#> \ 0) [0..<Suc \ T']]_{NL}, 1),$   
 $7 := ([\text{map } (\lambda t. \text{exc } zs \ t <\#\#> \ 1) [0..<Suc \ T']]_{NL}, 1),$   
 $18 := ([m']_N, 1),$   
 $19 := ([H]_N, 1),$   
 $20 := ([m' * H]_N, 1),$   
 $1 := \text{nlltape} ((\text{formula-}n \Phi_0 @ \text{formula-}n \Phi_1) @ \text{formula-}n \Phi_2),$   
 $42 := ([1]_N, 1),$   
 $43 := ([H]_N, 1),$   
 $44 := ([2]_N, 1),$   
 $50 := ([n]_N, 1)]$

**lemma** *tm15* [*transforms-intros*]:

**assumes**  $ttt = 114 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 9 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $6 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2$   
**shows** *transforms tm15 tps0 ttt tps15*  
 ⟨*proof*⟩

**definition** *tps16*  $\equiv$  *tpsD*

$[11 := ([n]_N, 1),$   
 $15 := ([p \ n]_N, 1),$   
 $16 := ([m]_N, 1),$   
 $17 := ([T']_N, 1),$   
 $4 := ([\text{map } (\lambda t. \text{exc } zs \ t <\#\#> \ 0) [0..<Suc \ T']]_{NL}, 1),$   
 $7 := ([\text{map } (\lambda t. \text{exc } zs \ t <\#\#> \ 1) [0..<Suc \ T']]_{NL}, 1),$   
 $18 := ([m']_N, 1),$   
 $19 := ([H]_N, 1),$   
 $20 := ([m' * H]_N, 1),$   
 $1 := \text{nlltape} ((\text{formula-}n \Phi_0 @ \text{formula-}n \Phi_1) @ \text{formula-}n \Phi_2),$   
 $42 := ([1]_N, 1),$   
 $43 := ([H]_N, 1),$   
 $44 := ([2]_N, 1),$   
 $50 := ([1 + 2 * n]_N, 1)]$

**lemma** *tm16* [*transforms-intros*]:

**assumes**  $ttt = 126 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 9 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $10 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2$   
**shows** *transforms tm16 tps0 ttt tps16*  
 ⟨*proof*⟩

**definition** *tps17*  $\equiv$  *tpsD*

$[11 := ([n]_N, 1),$   
 $15 := ([p \ n]_N, 1),$   
 $16 := ([m]_N, 1),$   
 $17 := ([T']_N, 1),$

$4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $18 := (\lfloor m' \rfloor_N, \ 1),$   
 $19 := (\lfloor H \rfloor_N, \ 1),$   
 $20 := (\lfloor m' * H \rfloor_N, \ 1),$   
 $1 := \text{nlltape } (\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3),$   
 $42 := (\lfloor 1 \rfloor_N, \ 1),$   
 $43 := (\lfloor H \rfloor_N, \ 1),$   
 $44 := (\lfloor 2 \rfloor_N, \ 1),$   
 $50 := (\lfloor 1 + 2 * n \rfloor_N, \ 1),$   
 $42 := (\lfloor 1 + 2 * n \rfloor_N, \ 1),$   
 $42 + 3 := (\lfloor 1 \rfloor_N, \ 1)$

**lemma** *tm17* [*transforms-intros*]:

**assumes**  $\text{ttt} = 126 + (2 * \text{d-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 9 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $10 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $\text{Suc } n * (9 + 1897 * (H^4 * (\text{nlength } (1 + 2 * n))^2))$   
**shows** *transforms tm17 tps0 ttt tps17*  
*<proof>*

**definition** *tpsE*  $\equiv$  *tpsD*

$42 := (\lfloor 1 \rfloor_N, \ 1),$   
 $43 := (\lfloor H \rfloor_N, \ 1),$   
 $44 := (\lfloor 2 \rfloor_N, \ 1),$   
 $50 := (\lfloor 1 + 2 * n \rfloor_N, \ 1),$   
 $42 := (\lfloor 1 + 2 * n \rfloor_N, \ 1),$   
 $42 + 3 := (\lfloor 1 \rfloor_N, \ 1)$

**lemma** *tpsE*:  $50 < j \implies j < 110 \implies \text{tpsE } ! j = (\lfloor \square \rfloor, \ 1)$   
*<proof>*

**lemma** *lentpsE*:  $\text{length } \text{tpsE} = 110$   
*<proof>*

**lemma** *tps17*:  $\text{tps17} = \text{tpsE}$

$11 := (\lfloor n \rfloor_N, \ 1),$   
 $15 := (\lfloor p \ n \rfloor_N, \ 1),$   
 $16 := (\lfloor m \rfloor_N, \ 1),$   
 $17 := (\lfloor T' \rfloor_N, \ 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $18 := (\lfloor m' \rfloor_N, \ 1),$   
 $19 := (\lfloor H \rfloor_N, \ 1),$   
 $20 := (\lfloor m' * H \rfloor_N, \ 1),$   
 $1 := \text{nlltape } (\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3)]$   
*<proof>*

**definition** *tps18*  $\equiv$  *tpsE*

$11 := (\lfloor n \rfloor_N, \ 1),$   
 $15 := (\lfloor p \ n \rfloor_N, \ 1),$   
 $16 := (\lfloor m \rfloor_N, \ 1),$   
 $17 := (\lfloor T' \rfloor_N, \ 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $18 := (\lfloor m' \rfloor_N, \ 1),$   
 $19 := (\lfloor H \rfloor_N, \ 1),$   
 $20 := (\lfloor m' * H \rfloor_N, \ 1),$   
 $1 := \text{nlltape } (\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3),$   
 $52 := (\lfloor H \rfloor_N, \ 1)$

**lemma** *tm18* [*transforms-intros*]:

**assumes**  $\text{ttt} = 136 + (2 * \text{d-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 11 * \text{nlength } H +$

$$5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$$

$$10 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$$

$$\text{Suc } n * (9 + 1897 * (H^4 * (\text{nlength } (1 + 2 * n))^2))$$
**shows** *transforms tm18 tps0 ttt tps18*  
 <proof>

**definition** *tps19*  $\equiv$  *tpsE*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } (\text{formula-}n \ \Phi_0 @ \text{formula-}n \ \Phi_1 @ \text{formula-}n \ \Phi_2 @ \text{formula-}n \ \Phi_3),$   
 $52 := (\lfloor H \rfloor_N, 1),$   
 $53 := (\lfloor 2 \rfloor_N, 1)]$

**lemma** *tm19* [*transforms-intros*]:

**assumes** *ttt*  $= 150 + (2 * \text{d-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 11 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $10 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $\text{Suc } n * (9 + 1897 * (H^4 * (\text{nlength } (1 + 2 * n))^2))$   
**shows** *transforms tm19 tps0 ttt tps19*  
 <proof>

**definition** *tps20*  $\equiv$  *tpsE*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } (\text{formula-}n \ \Phi_0 @ \text{formula-}n \ \Phi_1 @ \text{formula-}n \ \Phi_2 @ \text{formula-}n \ \Phi_3),$   
 $52 := (\lfloor H \rfloor_N, 1),$   
 $53 := (\lfloor 2 \rfloor_N, 1),$   
 $51 := (\lfloor n \rfloor_N, 1)]$

**lemma** *tm20* [*transforms-intros*]:

**assumes** *ttt*  $= 164 + (2 * \text{d-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 11 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $13 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $\text{Suc } n * (9 + 1897 * (H^4 * (\text{nlength } (1 + 2 * n))^2))$   
**shows** *transforms tm20 tps0 ttt tps20*  
 <proof>

**definition** *tps21*  $\equiv$  *tpsE*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } (\text{formula-}n \ \Phi_0 @ \text{formula-}n \ \Phi_1 @ \text{formula-}n \ \Phi_2 @ \text{formula-}n \ \Phi_3),$   
 $52 := (\lfloor H \rfloor_N, 1),$

$53 := (\lfloor 2 \rfloor_N, 1),$   
 $51 := (\lfloor 2 * n \rfloor_N, 1)]$

**lemma** *tm21* [*transforms-intros*]:

**assumes**  $ttt = 169 + (2 * d\text{-polynomial } p + 826) * (H + m')^{\wedge} 4 + 3 * nlength\ m' + 11 * nlength\ H +$   
 $5627 * H^{\wedge} 4 * (3 + nlength\ (3 * H + m' * H))^2 + 1875 * H^{\wedge} 4 +$   
 $15 * nlength\ n + 3764 * H^{\wedge} 4 * (3 + nlength\ (3 * H + 2 * n * H))^2 +$   
 $Suc\ n * (9 + 1897 * (H^{\wedge} 4 * (nlength\ (1 + 2 * n))^2))$

**shows** *transforms tm21 tps0 ttt tps21*  
 ⟨*proof*⟩

**definition** *tps22*  $\equiv$  *tpsE*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p\ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := nlltape\ (formula\text{-}n\ \Phi_0\ @\ formula\text{-}n\ \Phi_1\ @\ formula\text{-}n\ \Phi_2\ @\ formula\text{-}n\ \Phi_3),$   
 $52 := (\lfloor H \rfloor_N, 1),$   
 $53 := (\lfloor 2 \rfloor_N, 1),$   
 $51 := (\lfloor 2 * n + 3 \rfloor_N, 1)]$

**lemma** *tm22* [*transforms-intros*]:

**assumes**  $ttt = 184 + (2 * d\text{-polynomial } p + 826) * (H + m')^{\wedge} 4 + 3 * nlength\ m' + 11 * nlength\ H +$   
 $5627 * H^{\wedge} 4 * (3 + nlength\ (3 * H + m' * H))^2 + 1875 * H^{\wedge} 4 +$   
 $15 * nlength\ n + 3764 * H^{\wedge} 4 * (3 + nlength\ (3 * H + 2 * n * H))^2 +$   
 $Suc\ n * (9 + 1897 * (H^{\wedge} 4 * (nlength\ (1 + 2 * n))^2)) + 6 * nlength\ (2 * n + 3)$

**shows** *transforms tm22 tps0 ttt tps22*  
 ⟨*proof*⟩

**definition** *tps23*  $\equiv$  *tpsE*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p\ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := nlltape\ (formula\text{-}n\ \Phi_0\ @\ formula\text{-}n\ \Phi_1\ @\ formula\text{-}n\ \Phi_2\ @\ formula\text{-}n\ \Phi_3),$   
 $52 := (\lfloor H \rfloor_N, 1),$   
 $53 := (\lfloor 2 \rfloor_N, 1),$   
 $51 := (\lfloor 2 * n + 3 \rfloor_N, 1),$   
 $59 := (\lfloor m \rfloor_N, 1)]$

**lemma** *tm23* [*transforms-intros*]:

**assumes**  $ttt = 198 + (2 * d\text{-polynomial } p + 826) * (H + m')^{\wedge} 4 + 3 * nlength\ m' + 11 * nlength\ H +$   
 $5627 * H^{\wedge} 4 * (3 + nlength\ (3 * H + m' * H))^2 + 1875 * H^{\wedge} 4 +$   
 $15 * nlength\ n + 3764 * H^{\wedge} 4 * (3 + nlength\ (3 * H + 2 * n * H))^2 +$   
 $Suc\ n * (9 + 1897 * (H^{\wedge} 4 * (nlength\ (1 + 2 * n))^2)) + 6 * nlength\ (2 * n + 3) +$   
 $3 * nlength\ m$

**shows** *transforms tm23 tps0 ttt tps23*  
 ⟨*proof*⟩

**definition** *tps24*  $\equiv$  *tpsE*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p\ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$

$17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape (formula-n } \Phi_0 @ \text{formula-n } \Phi_1 @ \text{formula-n } \Phi_2 @ \text{formula-n } \Phi_3),$   
 $52 := (\lfloor H \rfloor_N, 1),$   
 $53 := (\lfloor 2 \rfloor_N, 1),$   
 $51 := (\lfloor 2 * n + 3 \rfloor_N, 1),$   
 $59 := (\lfloor Suc m \rfloor_N, 1)]$

**lemma** *tm24* [*transforms-intros*]:

**assumes**  $t t t = 203 + (2 * d\text{-polynomial } p + 826) * (H + m') ^ 4 + 3 * \text{nlength } m' + 11 * \text{nlength } H +$   
 $5627 * H ^ 4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H ^ 4 +$   
 $15 * \text{nlength } n + 3764 * H ^ 4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $Suc n * (9 + 1897 * (H ^ 4 * (\text{nlength } (1 + 2 * n))^2)) + 6 * \text{nlength } (2 * n + 3) +$   
 $5 * \text{nlength } m$

**shows** *transforms tm24 tps0 ttt tps24*

*<proof>*

**definition** *tps25*  $\equiv$  *tpsE*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape (formula-n } \Phi_0 @ \text{formula-n } \Phi_1 @ \text{formula-n } \Phi_2 @ \text{formula-n } \Phi_3 @ \text{formula-n } \Phi_4),$   
 $52 := (\lfloor H \rfloor_N, 1),$   
 $53 := (\lfloor 2 \rfloor_N, 1),$   
 $51 := (\lfloor 2 * n + 3 \rfloor_N, 1),$   
 $59 := (\lfloor Suc m \rfloor_N, 1),$   
 $51 := (\lfloor 2 * n + 2 + 1 + 2 * p n \rfloor_N, 1),$   
 $51 + 3 := (\lfloor 1 \rfloor_N, 1)]$

**lemma** *tm25* [*transforms-intros*]:

**assumes**  $t t t = 203 + (2 * d\text{-polynomial } p + 826) * (H + m') ^ 4 + 3 * \text{nlength } m' + 11 * \text{nlength } H +$   
 $5627 * H ^ 4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H ^ 4 +$   
 $15 * \text{nlength } n + 3764 * H ^ 4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $Suc n * (9 + 1897 * (H ^ 4 * (\text{nlength } (1 + 2 * n))^2)) + 6 * \text{nlength } (2 * n + 3) +$   
 $5 * \text{nlength } m + Suc (p n) * (9 + 1897 * (H ^ 4 * (\text{nlength } (Suc m))^2))$

**shows** *transforms tm25 tps0 ttt tps25*

*<proof>*

**definition** *tpsF*  $\equiv$  *tpsE*

$[52 := (\lfloor H \rfloor_N, 1),$   
 $53 := (\lfloor 2 \rfloor_N, 1),$   
 $51 := (\lfloor 2 * n + 3 \rfloor_N, 1),$   
 $59 := (\lfloor Suc m \rfloor_N, 1),$   
 $51 := (\lfloor 2 * n + 2 + 1 + 2 * p n \rfloor_N, 1),$   
 $51 + 3 := (\lfloor 1 \rfloor_N, 1)]$

**lemma** *tpsF*:  $59 < j \implies j < 110 \implies \text{tpsF } ! j = (\lfloor \square \rfloor, 1)$

*<proof>*

**lemma** *lentpsF*:  $\text{length } \text{tpsF} = 110$

*<proof>*

**lemma** *tps25*:  $\text{tps25} = \text{tpsF}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape (formula-n } \Phi_0 @ \text{formula-n } \Phi_1 @ \text{formula-n } \Phi_2 @ \text{formula-n } \Phi_3 @ \text{formula-n } \Phi_4)$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tps26} \equiv \text{tpsF}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape (formula-n } \Phi_0 @ \text{formula-n } \Phi_1 @ \text{formula-n } \Phi_2 @ \text{formula-n } \Phi_3 @ \text{formula-n } \Phi_4),$   
 $61 := (\lfloor H \rfloor_N, 1)]$

**lemma**  $\text{tm26}$  [*transforms-intros*]:

**assumes**  $\text{ttt} = 213 + (2 * \text{d-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 13 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $15 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $\text{Suc } n * (9 + 1897 * (H^4 * (\text{nlength } (1 + 2 * n))^2)) + 6 * \text{nlength } (2 * n + 3) +$   
 $5 * \text{nlength } m + \text{Suc } (p \ n) * (9 + 1897 * (H^4 * (\text{nlength } (\text{Suc } m))^2))$   
**shows**  $\text{transforms } \text{tm26 } \text{tps0 } \text{ttt } \text{tps26}$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tps27} \equiv \text{tpsF}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape (formula-n } \Phi_0 @ \text{formula-n } \Phi_1 @ \text{formula-n } \Phi_2 @ \text{formula-n } \Phi_3 @ \text{formula-n } \Phi_4),$   
 $61 := (\lfloor H \rfloor_N, 1),$   
 $60 := (\lfloor m \rfloor_N, 1)]$

**lemma**  $\text{tm27}$  [*transforms-intros*]:

**assumes**  $\text{ttt} = 227 + (2 * \text{d-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 13 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $15 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $\text{Suc } n * (9 + 1897 * (H^4 * (\text{nlength } (1 + 2 * n))^2)) + 6 * \text{nlength } (2 * n + 3) +$   
 $8 * \text{nlength } m + \text{Suc } (p \ n) * (9 + 1897 * (H^4 * (\text{nlength } (\text{Suc } m))^2))$   
**shows**  $\text{transforms } \text{tm27 } \text{tps0 } \text{ttt } \text{tps27}$   
 $\langle \text{proof} \rangle$

**definition**  $\text{tps28} \equiv \text{tpsF}$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$

$7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $18 := (\lfloor m' \rfloor_N, \ 1),$   
 $19 := (\lfloor H \rfloor_N, \ 1),$   
 $20 := (\lfloor m' * H \rfloor_N, \ 1),$   
 $1 := \text{nlldatape } (\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4),$   
 $61 := (\lfloor H \rfloor_N, \ 1),$   
 $60 := (\lfloor Suc \ m \rfloor_N, \ 1)]$

**lemma** *tm28* [*transforms-intros*]:

**assumes**  $ttt = 232 + (2 * d\text{-polynomial } p + 826) * (H + m')^{\wedge} 4 + 3 * \text{nlength } m' + 13 * \text{nlength } H +$   
 $5627 * H^{\wedge} 4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^{\wedge} 4 +$   
 $15 * \text{nlength } n + 3764 * H^{\wedge} 4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $Suc \ n * (9 + 1897 * (H^{\wedge} 4 * (\text{nlength } (1 + 2 * n))^2)) + 6 * \text{nlength } (2 * n + 3) +$   
 $10 * \text{nlength } m + Suc \ (p \ n) * (9 + 1897 * (H^{\wedge} 4 * (\text{nlength } (Suc \ m))^2))$

**shows** *transforms tm28 tps0 ttt tps28*

*<proof>*

**definition** *tps29*  $\equiv$  *tpsF*

$[11 := (\lfloor n \rfloor_N, \ 1),$   
 $15 := (\lfloor p \ n \rfloor_N, \ 1),$   
 $16 := (\lfloor m \rfloor_N, \ 1),$   
 $17 := (\lfloor T' \rfloor_N, \ 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $18 := (\lfloor m' \rfloor_N, \ 1),$   
 $19 := (\lfloor H \rfloor_N, \ 1),$   
 $20 := (\lfloor m' * H \rfloor_N, \ 1),$   
 $1 := \text{nlldatape } (\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4),$   
 $61 := (\lfloor H \rfloor_N, \ 1),$   
 $60 := (\lfloor Suc \ m \rfloor_N, \ 1),$   
 $68 := (\lfloor Suc \ m \rfloor_N, \ 1)]$

**lemma** *tm29* [*transforms-intros*]:

**assumes**  $ttt = 246 + (2 * d\text{-polynomial } p + 826) * (H + m')^{\wedge} 4 + 3 * \text{nlength } m' + 13 * \text{nlength } H +$   
 $5627 * H^{\wedge} 4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^{\wedge} 4 +$   
 $15 * \text{nlength } n + 3764 * H^{\wedge} 4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $Suc \ n * (9 + 1897 * (H^{\wedge} 4 * (\text{nlength } (1 + 2 * n))^2)) + 6 * \text{nlength } (2 * n + 3) +$   
 $10 * \text{nlength } m + Suc \ (p \ n) * (9 + 1897 * (H^{\wedge} 4 * (\text{nlength } (Suc \ m))^2)) +$   
 $3 * \text{nlength } (Suc \ m)$

**shows** *transforms tm29 tps0 ttt tps29*

*<proof>*

**definition** *tps30*  $\equiv$  *tpsF*

$[11 := (\lfloor n \rfloor_N, \ 1),$   
 $15 := (\lfloor p \ n \rfloor_N, \ 1),$   
 $16 := (\lfloor m \rfloor_N, \ 1),$   
 $17 := (\lfloor T' \rfloor_N, \ 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..<Suc \ T'] \rfloor_{NL}, \ 1),$   
 $18 := (\lfloor m' \rfloor_N, \ 1),$   
 $19 := (\lfloor H \rfloor_N, \ 1),$   
 $20 := (\lfloor m' * H \rfloor_N, \ 1),$   
 $1 := \text{nlldatape } (\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4),$   
 $61 := (\lfloor H \rfloor_N, \ 1),$   
 $60 := (\lfloor Suc \ m \rfloor_N, \ 1),$   
 $68 := (\lfloor T' + Suc \ m \rfloor_N, \ 1)]$

**lemma** *tm30* [*transforms-intros*]:

**assumes**  $ttt = 256 + (2 * d\text{-polynomial } p + 826) * (H + m')^{\wedge} 4 + 3 * \text{nlength } m' + 13 * \text{nlength } H +$   
 $5627 * H^{\wedge} 4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^{\wedge} 4 +$   
 $15 * \text{nlength } n + 3764 * H^{\wedge} 4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $Suc \ n * (9 + 1897 * (H^{\wedge} 4 * (\text{nlength } (1 + 2 * n))^2)) + 6 * \text{nlength } (2 * n + 3) +$   
 $10 * \text{nlength } m + Suc \ (p \ n) * (9 + 1897 * (H^{\wedge} 4 * (\text{nlength } (Suc \ m))^2)) +$

$3 * \text{nlength } (\text{Suc } m) + 3 * \max (\text{nlength } T') (\text{nlength } (\text{Suc } m))$   
**shows** *transforms tm30 tps0 ttt tps30*  
 <proof>

**definition** *tps31*  $\equiv$  *tpsF*

$[11 := ([n]_N, 1),$   
 $15 := ([p\ n]_N, 1),$   
 $16 := ([m]_N, 1),$   
 $17 := ([T']_N, 1),$   
 $4 := ([\text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<\text{Suc } T']_{NL}, 1),$   
 $7 := ([\text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<\text{Suc } T']_{NL}, 1),$   
 $18 := ([m']_N, 1),$   
 $19 := ([H]_N, 1),$   
 $20 := ([m' * H]_N, 1),$   
 $1 := \text{nlltape}$   
 $(\text{formula-}n\ \Phi_0\ @\ \text{formula-}n\ \Phi_1\ @\ \text{formula-}n\ \Phi_2\ @\ \text{formula-}n\ \Phi_3\ @\ \text{formula-}n\ \Phi_4\ @$   
 $\text{formula-}n\ \Phi_5),$   
 $61 := ([H]_N, 1),$   
 $60 := ([\text{Suc } m]_N, 1),$   
 $68 := ([T' + \text{Suc } m]_N, 1),$   
 $60 := ([\text{Suc } m + T']_N, 1),$   
 $60 + 3 := ([1]_N, 1)]$

**definition** *ttt31*  $\equiv$   $256 + (2 * d\text{-polynomial } p + 826) * (H + m')^4 + 3 * \text{nlength } m' + 13 * \text{nlength } H +$   
 $5627 * H^4 * (3 + \text{nlength } (3 * H + m' * H))^2 + 1875 * H^4 +$   
 $15 * \text{nlength } n + 3764 * H^4 * (3 + \text{nlength } (3 * H + 2 * n * H))^2 +$   
 $\text{Suc } n * (9 + 1897 * (H^4 * (\text{nlength } (1 + 2 * n))^2)) + 6 * \text{nlength } (2 * n + 3) +$   
 $10 * \text{nlength } m + \text{Suc } (p\ n) * (9 + 1897 * (H^4 * (\text{nlength } (\text{Suc } m))^2)) +$   
 $3 * \text{nlength } (\text{Suc } m) + 3 * \max (\text{nlength } T') (\text{nlength } (\text{Suc } m)) +$   
 $\text{Suc } T' * (9 + 1891 * (H^4 * (\text{nlength } (\text{Suc } m + T'))^2))$

**lemma** *le-N*:  $y \leq 2 * n + 2 * p\ n + 3 + T' \implies y \leq N$   
 <proof>

**lemma** *n-le-N*:  $n \leq N$   
 <proof>

**lemma** *H-le-N*:  $H \leq N$   
 <proof>

**lemma** *N-ge-1*:  $N \geq 1$   
 <proof>

**lemma** *pow2-sum-le*:  
**fixes**  $a\ b :: \text{nat}$   
**shows**  $(a + b)^2 \leq a^2 + (2 * a + 1) * b^2$   
 <proof>

**lemma** *ttt31*:  $\text{ttt31} \leq (32 * d\text{-polynomial } p + 222011) * H^4 * N^4$   
 <proof>

**lemma** *tm31* [*transforms-intros*]: *transforms tm31 tps0 ttt31 tps31*  
 <proof>

**definition** *tpsG*  $\equiv$  *tpsF*

$[61 := ([H]_N, 1),$   
 $60 := ([\text{Suc } m]_N, 1),$   
 $68 := ([T' + \text{Suc } m]_N, 1),$   
 $60 := ([\text{Suc } m + T']_N, 1),$   
 $60 + 3 := ([1]_N, 1)]$

**lemma** *tpsG*:  $68 < j \implies j < 110 \implies \text{tpsG } !\ j = ([\square], 1)$   
 <proof>

**lemma** *lentpsG*: *length tpsG = 110*

*<proof>*

**lemma** *tps31*: *tps31 = tpsG*

*[11 := ([n]<sub>N</sub>, 1),*

*15 := ([p n]<sub>N</sub>, 1),*

*16 := ([m]<sub>N</sub>, 1),*

*17 := ([T']<sub>N</sub>, 1),*

*4 := ([map (λt. exc zs t <#> 0) [0..<Suc T']]<sub>NL</sub>, 1),*

*7 := ([map (λt. exc zs t <#> 1) [0..<Suc T']]<sub>NL</sub>, 1),*

*18 := ([m']<sub>N</sub>, 1),*

*19 := ([H]<sub>N</sub>, 1),*

*20 := ([m' \* H]<sub>N</sub>, 1),*

*1 := nlltape*

*(formula-n Φ<sub>0</sub> @ formula-n Φ<sub>1</sub> @ formula-n Φ<sub>2</sub> @ formula-n Φ<sub>3</sub> @ formula-n Φ<sub>4</sub> @*  
*formula-n Φ<sub>5</sub>)*

*<proof>*

**definition** *tps32* ≡ *tpsG*

*[11 := ([n]<sub>N</sub>, 1),*

*15 := ([p n]<sub>N</sub>, 1),*

*16 := ([m]<sub>N</sub>, 1),*

*17 := ([T']<sub>N</sub>, 1),*

*4 := ([map (λt. exc zs t <#> 0) [0..<Suc T']]<sub>NL</sub>, 1),*

*7 := ([map (λt. exc zs t <#> 1) [0..<Suc T']]<sub>NL</sub>, 1),*

*18 := ([m']<sub>N</sub>, 1),*

*19 := ([H]<sub>N</sub>, 1),*

*20 := ([m' \* H]<sub>N</sub>, 1),*

*1 := nlltape*

*(formula-n Φ<sub>0</sub> @ formula-n Φ<sub>1</sub> @ formula-n Φ<sub>2</sub> @ formula-n Φ<sub>3</sub> @ formula-n Φ<sub>4</sub> @*  
*formula-n Φ<sub>5</sub>),*

*69 := ([2]<sub>N</sub>, 1)]*

**lemma** *tm32* [*transforms-intros*]:

**assumes** *ttt = ttt31 + 14*

**shows** *transforms tm32 tps0 ttt tps32*

*<proof>*

**definition** *tps33* ≡ *tpsG*

*[11 := ([n]<sub>N</sub>, 1),*

*15 := ([p n]<sub>N</sub>, 1),*

*16 := ([m]<sub>N</sub>, 1),*

*17 := ([T']<sub>N</sub>, 1),*

*4 := ([map (λt. exc zs t <#> 0) [0..<Suc T']]<sub>NL</sub>, 1),*

*7 := ([map (λt. exc zs t <#> 1) [0..<Suc T']]<sub>NL</sub>, 1),*

*18 := ([m']<sub>N</sub>, 1),*

*19 := ([H]<sub>N</sub>, 1),*

*20 := ([m' \* H]<sub>N</sub>, 1),*

*1 := nlltape*

*(formula-n Φ<sub>0</sub> @ formula-n Φ<sub>1</sub> @ formula-n Φ<sub>2</sub> @ formula-n Φ<sub>3</sub> @ formula-n Φ<sub>4</sub> @*  
*formula-n Φ<sub>5</sub>),*

*69 := ([2]<sub>N</sub>, 1),*

*70 := ([H]<sub>N</sub>, 1)]*

**lemma** *tm33* [*transforms-intros*]:

**assumes** *ttt = ttt31 + 24 + 2 \* nlength H*

**shows** *transforms tm33 tps0 ttt tps33*

*<proof>*

**definition** *tps34* ≡ *tpsG*

*[11 := ([n]<sub>N</sub>, 1),*

*15 := ([p n]<sub>N</sub>, 1),*

$16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{lltape}$   
 $(\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4 \ @$   
 $\text{formula-}n \ \Phi_5 \ @ \ \text{formula-}n \ \Phi_6),$   
 $69 := (\lfloor 2 \rfloor_N, 1),$   
 $70 := (\lfloor H \rfloor_N, 1),$   
 $0 := (\lfloor xs \rfloor, \text{Suc } n),$   
 $69 := (\lfloor 2 + 2 * n \rfloor_N, 1)]$

**lemma** *tm34* [*transforms-intros*]:

**assumes**  $ttt = ttt31 + 24 + 2 * \text{nlength } H + (133650 * H \wedge 6 * n \wedge 3 + 1)$

**shows** *transforms tm34 tps0 ttt tps34*

*<proof>*

**definition** *tpsH*  $\equiv$  *tpsG*

$69 := (\lfloor 2 \rfloor_N, 1),$   
 $70 := (\lfloor H \rfloor_N, 1),$   
 $0 := (\lfloor xs \rfloor, \text{Suc } n),$   
 $69 := (\lfloor 2 + 2 * n \rfloor_N, 1)]$

**lemma** *tpsH*:  $76 < j \implies j < 110 \implies \text{tpsH } ! j = (\lfloor [] \rfloor, 1)$

*<proof>*

**lemma** *lentpsH*:  $\text{length } \text{tpsH} = 110$

*<proof>*

**lemma** *tps34*:  $\text{tps34} = \text{tpsH}$

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{lltape}$   
 $(\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4 \ @$   
 $\text{formula-}n \ \Phi_5 \ @ \ \text{formula-}n \ \Phi_6)]$   
*<proof>*

**definition** *tps35*  $\equiv$  *tpsH*

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{lltape}$   
 $(\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4 \ @$   
 $\text{formula-}n \ \Phi_5 \ @ \ \text{formula-}n \ \Phi_6),$   
 $77 := (\lfloor n \rfloor_N, 1)]$

**lemma** *tm35* [*transforms-intros*]:

**assumes**  $t_{tt} = t_{tt}31 + 38 + 2 * nlength\ H + (133650 * H^6 * n^3 + 1) + 3 * nlength\ n$   
**shows** *transforms tm35 tps0 ttt tps35*  
 ⟨proof⟩

**definition**  $tps36 \equiv tpsH$

[11 := ( $\lfloor n \rfloor_N, 1$ ),  
 15 := ( $\lfloor p\ n \rfloor_N, 1$ ),  
 16 := ( $\lfloor m \rfloor_N, 1$ ),  
 17 := ( $\lfloor T' \rfloor_N, 1$ ),  
 4 := ( $\lfloor map\ (\lambda t.\ exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T']_{NL}, 1$ ),  
 7 := ( $\lfloor map\ (\lambda t.\ exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T']_{NL}, 1$ ),  
 18 := ( $\lfloor m' \rfloor_N, 1$ ),  
 19 := ( $\lfloor H \rfloor_N, 1$ ),  
 20 := ( $\lfloor m' * H \rfloor_N, 1$ ),  
 1 := *nlldatape*  
 (formula- $n\ \Phi_0$  @ formula- $n\ \Phi_1$  @ formula- $n\ \Phi_2$  @ formula- $n\ \Phi_3$  @ formula- $n\ \Phi_4$  @  
 formula- $n\ \Phi_5$  @ formula- $n\ \Phi_6$ ),  
 77 := ( $\lfloor 2 * n \rfloor_N, 1$ )

**lemma**  $tm36$  [*transforms-intros*]:

**assumes**  $t_{tt} = t_{tt}31 + 43 + 2 * nlength\ H + (133650 * H^6 * n^3 + 1) + 5 * nlength\ n$   
**shows** *transforms tm36 tps0 ttt tps36*  
 ⟨proof⟩

**definition**  $tps37 \equiv tpsH$

[11 := ( $\lfloor n \rfloor_N, 1$ ),  
 15 := ( $\lfloor p\ n \rfloor_N, 1$ ),  
 16 := ( $\lfloor m \rfloor_N, 1$ ),  
 17 := ( $\lfloor T' \rfloor_N, 1$ ),  
 4 := ( $\lfloor map\ (\lambda t.\ exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T']_{NL}, 1$ ),  
 7 := ( $\lfloor map\ (\lambda t.\ exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T']_{NL}, 1$ ),  
 18 := ( $\lfloor m' \rfloor_N, 1$ ),  
 19 := ( $\lfloor H \rfloor_N, 1$ ),  
 20 := ( $\lfloor m' * H \rfloor_N, 1$ ),  
 1 := *nlldatape*  
 (formula- $n\ \Phi_0$  @ formula- $n\ \Phi_1$  @ formula- $n\ \Phi_2$  @ formula- $n\ \Phi_3$  @ formula- $n\ \Phi_4$  @  
 formula- $n\ \Phi_5$  @ formula- $n\ \Phi_6$ ),  
 77 := ( $\lfloor 2 * n + 4 \rfloor_N, 1$ )

**lemma**  $tm37$  [*transforms-intros*]:

**assumes**  $t_{tt} = t_{tt}31 + 63 + 2 * nlength\ H + (133650 * H^6 * n^3 + 1) + 5 * nlength\ n +$   
 $8 * nlength\ (2 * n + 4)$   
**shows** *transforms tm37 tps0 ttt tps37*  
 ⟨proof⟩

**definition**  $tps38 \equiv tpsH$

[11 := ( $\lfloor n \rfloor_N, 1$ ),  
 15 := ( $\lfloor p\ n \rfloor_N, 1$ ),  
 16 := ( $\lfloor m \rfloor_N, 1$ ),  
 17 := ( $\lfloor T' \rfloor_N, 1$ ),  
 4 := ( $\lfloor map\ (\lambda t.\ exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T']_{NL}, 1$ ),  
 7 := ( $\lfloor map\ (\lambda t.\ exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T']_{NL}, 1$ ),  
 18 := ( $\lfloor m' \rfloor_N, 1$ ),  
 19 := ( $\lfloor H \rfloor_N, 1$ ),  
 20 := ( $\lfloor m' * H \rfloor_N, 1$ ),  
 1 := *nlldatape*  
 (formula- $n\ \Phi_0$  @ formula- $n\ \Phi_1$  @ formula- $n\ \Phi_2$  @ formula- $n\ \Phi_3$  @ formula- $n\ \Phi_4$  @  
 formula- $n\ \Phi_5$  @ formula- $n\ \Phi_6$ ),  
 77 := ( $\lfloor 2 * n + 4 \rfloor_N, 1$ ),  
 78 := ( $\lfloor H \rfloor_N, 1$ )

**lemma**  $tm38$  [*transforms-intros*]:

**assumes**  $t_{tt} = t_{tt}31 + 73 + 2 * nlength\ H + (133650 * H^6 * n^3 + 1) +$



**lemma** *lentpsI*:  $\text{length } tpsI = 110$

*<proof>*

**lemma** *tps40*:  $tps40 = tpsI$

[11 := ( $\lfloor n \rfloor_N, 1$ ),

15 := ( $\lfloor p \ n \rfloor_N, 1$ ),

16 := ( $\lfloor m \rfloor_N, 1$ ),

17 := ( $\lfloor T' \rfloor_N, 1$ ),

4 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),

7 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),

18 := ( $\lfloor m' \rfloor_N, 1$ ),

19 := ( $\lfloor H \rfloor_N, 1$ ),

20 := ( $\lfloor m' * H \rfloor_N, 1$ ),

1 := *nlldatape*

(*formula-n*  $\Phi_0$  @ *formula-n*  $\Phi_1$  @ *formula-n*  $\Phi_2$  @ *formula-n*  $\Phi_3$  @ *formula-n*  $\Phi_4$  @  
*formula-n*  $\Phi_5$  @ *formula-n*  $\Phi_6$  @ *formula-n*  $\Phi_7$ )

*<proof>*

**definition** *tps41*  $\equiv$  *tpsI*

[11 := ( $\lfloor n \rfloor_N, 1$ ),

15 := ( $\lfloor p \ n \rfloor_N, 1$ ),

16 := ( $\lfloor m \rfloor_N, 1$ ),

17 := ( $\lfloor T' \rfloor_N, 1$ ),

4 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),

7 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),

18 := ( $\lfloor m' \rfloor_N, 1$ ),

19 := ( $\lfloor H \rfloor_N, 1$ ),

20 := ( $\lfloor m' * H \rfloor_N, 1$ ),

1 := *nlldatape*

(*formula-n*  $\Phi_0$  @ *formula-n*  $\Phi_1$  @ *formula-n*  $\Phi_2$  @ *formula-n*  $\Phi_3$  @ *formula-n*  $\Phi_4$  @  
*formula-n*  $\Phi_5$  @ *formula-n*  $\Phi_6$  @ *formula-n*  $\Phi_7$ ),

84 := ( $\lfloor m' \rfloor_N, 1$ )

**lemma** *tm41* [*transforms-intros*]:

**assumes**  $ttt = ttt31 + 102 + 2 * \text{nlength } H + (133650 * H^6 * n^3 + 1) + 5 * \text{nlength } n +$   
 $8 * \text{nlength } (2 * n + 4) + 2 * \text{nlength } H + 3 * \text{nlength } (p \ n) +$   
 $p \ n * 257 * H * (\text{nlength } (2 * n + 4 + 2 * p \ n) + \text{nlength } H)^2 +$   
 $3 * \text{nlength } m'$

**shows** *transforms* *tm41* *tps0* *ttt* *tps41*

*<proof>*

**definition** *tps42*  $\equiv$  *tpsI*

[11 := ( $\lfloor n \rfloor_N, 1$ ),

15 := ( $\lfloor p \ n \rfloor_N, 1$ ),

16 := ( $\lfloor m \rfloor_N, 1$ ),

17 := ( $\lfloor T' \rfloor_N, 1$ ),

4 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),

7 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),

18 := ( $\lfloor m' \rfloor_N, 1$ ),

19 := ( $\lfloor H \rfloor_N, 1$ ),

20 := ( $\lfloor m' * H \rfloor_N, 1$ ),

1 := *nlldatape*

(*formula-n*  $\Phi_0$  @ *formula-n*  $\Phi_1$  @ *formula-n*  $\Phi_2$  @ *formula-n*  $\Phi_3$  @ *formula-n*  $\Phi_4$  @  
*formula-n*  $\Phi_5$  @ *formula-n*  $\Phi_6$  @ *formula-n*  $\Phi_7$ ),

84 := ( $\lfloor T' + m' \rfloor_N, 1$ )

**lemma** *tm42* [*transforms-intros*]:

**assumes**  $ttt = ttt31 + 112 + 2 * \text{nlength } H + (133650 * H^6 * n^3 + 1) + 5 * \text{nlength } n +$   
 $8 * \text{nlength } (2 * n + 4) + 2 * \text{nlength } H + 3 * \text{nlength } (p \ n) +$   
 $p \ n * 257 * H * (\text{nlength } (2 * n + 4 + 2 * p \ n) + \text{nlength } H)^2 + 3 * \text{nlength } m' +$   
 $3 * \max (\text{nlength } T') (\text{nlength } m')$

**shows** *transforms* *tm42* *tps0* *ttt* *tps42*

*<proof>*

**definition**  $tps43 \equiv tpsI$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{niltape}$   
 $(\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4 \ @$   
 $\text{formula-}n \ \Phi_5 \ @ \ \text{formula-}n \ \Phi_6 \ @ \ \text{formula-}n \ \Phi_7),$   
 $84 := (\lfloor 2 * T' + m' \rfloor_N, 1)]$

**lemma**  $tm43$  [*transforms-intros*]:

**assumes**  $ttt = ttt31 + 122 + 2 * \text{nlength } H + (133650 * H^6 * n^3 + 1) + 5 * \text{nlength } n +$   
 $8 * \text{nlength } (2 * n + 4) + 2 * \text{nlength } H + 3 * \text{nlength } (p \ n) +$   
 $p \ n * 257 * H * (\text{nlength } (2 * n + 4 + 2 * p \ n) + \text{nlength } H)^2 + 3 * \text{nlength } m' +$   
 $3 * \max (\text{nlength } T') (\text{nlength } m') +$   
 $3 * \max (\text{nlength } T') (\text{nlength } (T' + m'))$   
**shows**  $\text{transforms } tm43 \ tps0 \ ttt \ tps43$   
*<proof>*

**definition**  $tps44 \equiv tpsI$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{niltape}$   
 $(\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4 \ @$   
 $\text{formula-}n \ \Phi_5 \ @ \ \text{formula-}n \ \Phi_6 \ @ \ \text{formula-}n \ \Phi_7),$   
 $84 := (\lfloor 3 * T' + m' \rfloor_N, 1)]$

**lemma**  $tm44$  [*transforms-intros*]:

**assumes**  $ttt = ttt31 + 132 + 2 * \text{nlength } H + (133650 * H^6 * n^3 + 1) + 5 * \text{nlength } n +$   
 $8 * \text{nlength } (2 * n + 4) + 2 * \text{nlength } H + 3 * \text{nlength } (p \ n) +$   
 $p \ n * 257 * H * (\text{nlength } (2 * n + 4 + 2 * p \ n) + \text{nlength } H)^2 + 3 * \text{nlength } m' +$   
 $3 * \max (\text{nlength } T') (\text{nlength } m') +$   
 $3 * \max (\text{nlength } T') (\text{nlength } (T' + m')) +$   
 $3 * \max (\text{nlength } T') (\text{nlength } (2 * T' + m'))$   
**shows**  $\text{transforms } tm44 \ tps0 \ ttt \ tps44$   
*<proof>*

**definition**  $tps45 \equiv tpsI$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{niltape}$   
 $(\text{formula-}n \ \Phi_0 \ @ \ \text{formula-}n \ \Phi_1 \ @ \ \text{formula-}n \ \Phi_2 \ @ \ \text{formula-}n \ \Phi_3 \ @ \ \text{formula-}n \ \Phi_4 \ @$   
 $\text{formula-}n \ \Phi_5 \ @ \ \text{formula-}n \ \Phi_6 \ @ \ \text{formula-}n \ \Phi_7),$

$84 := (\lfloor 1 + 3 * T' + m' \rfloor_N, 1)$

**lemma** *tm45* [*transforms-intros*]:

**assumes**  $ttt = ttt31 + 137 + 2 * nlength\ H + (133650 * H \wedge 6 * n \wedge 3 + 1) + 5 * nlength\ n +$   
 $8 * nlength\ (2 * n + 4) + 2 * nlength\ H + 3 * nlength\ (p\ n) +$   
 $p\ n * 257 * H * (nlength\ (2 * n + 4 + 2 * p\ n) + nlength\ H)^2 + 3 * nlength\ m' +$   
 $3 * max\ (nlength\ T')\ (nlength\ m') +$   
 $3 * max\ (nlength\ T')\ (nlength\ (T' + m')) +$   
 $3 * max\ (nlength\ T')\ (nlength\ (2 * T' + m')) +$   
 $2 * nlength\ (3 * T' + m')$   
**shows** *transforms tm45 tps0 ttt tps45*  
*<proof>*

**definition** *tps46*  $\equiv$  *tpsI*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p\ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := nlltape$   
 $(formula-n\ \Phi_0 @ formula-n\ \Phi_1 @ formula-n\ \Phi_2 @ formula-n\ \Phi_3 @ formula-n\ \Phi_4 @$   
 $formula-n\ \Phi_5 @ formula-n\ \Phi_6 @ formula-n\ \Phi_7),$   
 $84 := (\lfloor 1 + 3 * T' + m' \rfloor_N, 1),$   
 $85 := (\lfloor H \rfloor_N, 1)]$

**lemma** *tm46* [*transforms-intros*]:

**assumes**  $ttt = ttt31 + 147 + 2 * nlength\ H + (133650 * H \wedge 6 * n \wedge 3 + 1) + 5 * nlength\ n +$   
 $8 * nlength\ (2 * n + 4) + 4 * nlength\ H + 3 * nlength\ (p\ n) +$   
 $p\ n * 257 * H * (nlength\ (2 * n + 4 + 2 * p\ n) + nlength\ H)^2 + 3 * nlength\ m' +$   
 $3 * max\ (nlength\ T')\ (nlength\ m') +$   
 $3 * max\ (nlength\ T')\ (nlength\ (T' + m')) +$   
 $3 * max\ (nlength\ T')\ (nlength\ (2 * T' + m')) +$   
 $2 * nlength\ (3 * T' + m')$   
**shows** *transforms tm46 tps0 ttt tps46*  
*<proof>*

**definition** *tps47*  $\equiv$  *tpsI*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p\ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 0)\ [0..<Suc\ T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor map\ (\lambda t. exc\ zs\ t\ <\#\>\ 1)\ [0..<Suc\ T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := nlltape$   
 $(formula-n\ \Phi_0 @ formula-n\ \Phi_1 @ formula-n\ \Phi_2 @ formula-n\ \Phi_3 @ formula-n\ \Phi_4 @$   
 $formula-n\ \Phi_5 @ formula-n\ \Phi_6 @ formula-n\ \Phi_7 @ formula-n\ \Phi_8),$   
 $84 := (\lfloor 1 + 3 * T' + m' \rfloor_N, 1),$   
 $85 := (\lfloor H \rfloor_N, 1),$   
 $84 + 2 := (\lfloor 3 \rfloor_N, 1),$   
 $84 + 6 := (\lfloor formula-n\ \Phi_8 \rfloor_{NLL}, 1)]$

**definition** *ttt47*  $\equiv$  *ttt31* + 166 +

$6 * nlength\ H +$   
 $133650 * H \wedge 6 * n \wedge 3 +$   
 $5 * nlength\ n +$   
 $8 * nlength\ (2 * n + 4) +$

$$\begin{aligned}
& 3 * \text{nlength } (p \ n) + \\
& p \ n * 257 * H * (\text{nlength } (2 * n + 4 + 2 * p \ n) + \text{nlength } H)^2 + \\
& 3 * \text{nlength } m' + \\
& 3 * \max (\text{nlength } T') (\text{nlength } m') + \\
& 3 * \max (\text{nlength } T') (\text{nlength } (T' + m')) + \\
& 3 * \max (\text{nlength } T') (\text{nlength } (2 * T' + m')) + \\
& 2 * \text{nlength } (3 * T' + m') + \\
& 1861 * H^4 * (\text{nlength } (\text{Suc } (1 + 3 * T' + m')))^2
\end{aligned}$$

**lemma** *ttt47*:  $t_{tt47} \leq (32 * d\text{-polynomial } p + 364343) * H^6 * N^4$   
<proof>

**lemma** *tm47* [*transforms-intros*]: *transforms tm47 tps0 ttt47 tps47*  
<proof>

**definition** *tpsJ*  $\equiv$  *tpsI*

$$\begin{aligned}
84 & := (\lfloor 1 + 3 * T' + m' \rfloor_N, 1), \\
85 & := (\lfloor H \rfloor_N, 1), \\
84 + 2 & := (\lfloor 3 \rfloor_N, 1), \\
84 + 6 & := (\lfloor \text{formula-}n \ \Phi_8 \rfloor_{NLL}, 1)
\end{aligned}$$

**lemma** *tpsJ*:  $90 < j \implies j < 110 \implies t_{psJ} ! j = (\lfloor \rfloor, 1)$   
<proof>

**lemma** *lentpsJ*:  $\text{length } t_{psJ} = 110$   
<proof>

**lemma** *tps47*:  $t_{ps47} = t_{psJ}$

$$\begin{aligned}
11 & := (\lfloor n \rfloor_N, 1), \\
15 & := (\lfloor p \ n \rfloor_N, 1), \\
16 & := (\lfloor m \rfloor_N, 1), \\
17 & := (\lfloor T' \rfloor_N, 1), \\
4 & := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<\text{Suc } T'] \rfloor_{NLL}, 1), \\
7 & := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<\text{Suc } T'] \rfloor_{NLL}, 1), \\
18 & := (\lfloor m' \rfloor_N, 1), \\
19 & := (\lfloor H \rfloor_N, 1), \\
20 & := (\lfloor m' * H \rfloor_N, 1), \\
1 & := \text{niltape} \\
& (\text{formula-}n \ \Phi_0 @ \text{formula-}n \ \Phi_1 @ \text{formula-}n \ \Phi_2 @ \text{formula-}n \ \Phi_3 @ \text{formula-}n \ \Phi_4 @ \\
& \text{formula-}n \ \Phi_5 @ \text{formula-}n \ \Phi_6 @ \text{formula-}n \ \Phi_7 @ \text{formula-}n \ \Phi_8)
\end{aligned}$$

<proof>

**definition** *tps48*  $\equiv$  *tpsJ*

$$\begin{aligned}
11 & := (\lfloor n \rfloor_N, 1), \\
15 & := (\lfloor p \ n \rfloor_N, 1), \\
16 & := (\lfloor m \rfloor_N, 1), \\
17 & := (\lfloor T' \rfloor_N, 1), \\
4 & := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<\text{Suc } T'] \rfloor_{NLL}, 1), \\
7 & := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<\text{Suc } T'] \rfloor_{NLL}, 1), \\
18 & := (\lfloor m' \rfloor_N, 1), \\
19 & := (\lfloor H \rfloor_N, 1), \\
20 & := (\lfloor m' * H \rfloor_N, 1), \\
1 & := \text{niltape} \\
& (\text{formula-}n \ \Phi_0 @ \text{formula-}n \ \Phi_1 @ \text{formula-}n \ \Phi_2 @ \text{formula-}n \ \Phi_3 @ \text{formula-}n \ \Phi_4 @ \\
& \text{formula-}n \ \Phi_5 @ \text{formula-}n \ \Phi_6 @ \text{formula-}n \ \Phi_7 @ \text{formula-}n \ \Phi_8), \\
91 & := (\lfloor N \rfloor_N, 1)
\end{aligned}$$

**lemma** *tm48* [*transforms-intros*]:  
**assumes**  $t_{tt} = t_{tt47} + 14 + 3 * \text{nlength } N$   
**shows** *transforms tm48 tps0 ttt tps48*  
<proof>

**definition** *tps49*  $\equiv$  *tpsJ*

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc\ T'\rfloor]_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc\ T'\rfloor]_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{lltape}$   
 $(\text{formula-}n\ \Phi_0 @ \text{formula-}n\ \Phi_1 @ \text{formula-}n\ \Phi_2 @ \text{formula-}n\ \Phi_3 @ \text{formula-}n\ \Phi_4 @$   
 $\text{formula-}n\ \Phi_5 @ \text{formula-}n\ \Phi_6 @ \text{formula-}n\ \Phi_7 @ \text{formula-}n\ \Phi_8),$   
 $91 := (\lfloor N \rfloor_N, 1),$   
 $92 := (\lfloor H \rfloor_N, 1)]$

**lemma** *tm49* [*transforms-intros*]:

**assumes**  $ttt = ttt47 + 24 + 3 * nlength\ N + 2 * nlength\ H$   
**shows** *transforms tm49 tps0 ttt tps49*  
*<proof>*

**definition** *tps50*  $\equiv$  *tpsJ*

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc\ T'\rfloor]_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc\ T'\rfloor]_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{lltape}$   
 $(\text{formula-}n\ \Phi_0 @ \text{formula-}n\ \Phi_1 @ \text{formula-}n\ \Phi_2 @ \text{formula-}n\ \Phi_3 @ \text{formula-}n\ \Phi_4 @$   
 $\text{formula-}n\ \Phi_5 @ \text{formula-}n\ \Phi_6 @ \text{formula-}n\ \Phi_7 @ \text{formula-}n\ \Phi_8),$   
 $91 := (\lfloor N \rfloor_N, 1),$   
 $92 := (\lfloor H \rfloor_N, 1),$   
 $93 := (\lfloor Z \rfloor_N, 1)]$

**lemma** *tm50* [*transforms-intros*]:

**assumes**  $ttt = ttt47 + 34 + 3 * nlength\ N + 2 * nlength\ H + 2 * nlength\ Z$   
**shows** *transforms tm50 tps0 ttt tps50*  
*<proof>*

**definition** *tps51*  $\equiv$  *tpsJ*

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc\ T'\rfloor]_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc\ T'\rfloor]_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{lltape}$   
 $(\text{formula-}n\ \Phi_0 @ \text{formula-}n\ \Phi_1 @ \text{formula-}n\ \Phi_2 @ \text{formula-}n\ \Phi_3 @ \text{formula-}n\ \Phi_4 @$   
 $\text{formula-}n\ \Phi_5 @ \text{formula-}n\ \Phi_6 @ \text{formula-}n\ \Phi_7 @ \text{formula-}n\ \Phi_8),$   
 $91 := (\lfloor N \rfloor_N, 1),$   
 $92 := (\lfloor H \rfloor_N, 1),$   
 $93 := (\lfloor Z \rfloor_N, 1),$   
 $94 := (\lfloor T' \rfloor_N, 1)]$

**lemma** *tm51* [*transforms-intros*]:

**assumes**  $ttt = ttt47 + 48 + 3 * nlength\ N + 2 * nlength\ H + 2 * nlength\ Z + 3 * nlength\ T'$   
**shows** *transforms tm51 tps0 ttt tps51*

*<proof>*

**definition** *tps52*  $\equiv$  *tpsJ*

[11 := ( $\lfloor n \rfloor_N, 1$ ),  
15 := ( $\lfloor p \ n \rfloor_N, 1$ ),  
16 := ( $\lfloor m \rfloor_N, 1$ ),  
17 := ( $\lfloor T' \rfloor_N, 1$ ),  
4 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),  
7 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),  
18 := ( $\lfloor m' \rfloor_N, 1$ ),  
19 := ( $\lfloor H \rfloor_N, 1$ ),  
20 := ( $\lfloor m' * H \rfloor_N, 1$ ),  
1 := *nlltape*  
    (*formula-n*  $\Phi_0$  @ *formula-n*  $\Phi_1$  @ *formula-n*  $\Phi_2$  @ *formula-n*  $\Phi_3$  @ *formula-n*  $\Phi_4$  @  
    *formula-n*  $\Phi_5$  @ *formula-n*  $\Phi_6$  @ *formula-n*  $\Phi_7$  @ *formula-n*  $\Phi_8$ ),  
91 := ( $\lfloor N \rfloor_N, 1$ ),  
92 := ( $\lfloor H \rfloor_N, 1$ ),  
93 := ( $\lfloor Z \rfloor_N, 1$ ),  
94 := ( $\lfloor T' \rfloor_N, 1$ ),  
95 := ( $\lfloor \text{formula-n } \psi \rfloor_{NLL}, 1$ )

**lemma** *tm52* [*transforms-intros*]:

**assumes**  $ttt = ttt47 + 58 + 3 * \text{nlength } N + 2 * \text{nlength } H + 2 * \text{nlength } Z +$   
     $3 * \text{nlength } T' + 2 * \text{nlllength } (\text{formula-n } \psi)$   
**shows** *transforms tm52 tps0 ttt tps52*  
*<proof>*

**definition** *tps53*  $\equiv$  *tpsJ*

[11 := ( $\lfloor n \rfloor_N, 1$ ),  
15 := ( $\lfloor p \ n \rfloor_N, 1$ ),  
16 := ( $\lfloor m \rfloor_N, 1$ ),  
17 := ( $\lfloor T' \rfloor_N, 1$ ),  
4 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),  
7 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),  
18 := ( $\lfloor m' \rfloor_N, 1$ ),  
19 := ( $\lfloor H \rfloor_N, 1$ ),  
20 := ( $\lfloor m' * H \rfloor_N, 1$ ),  
1 := *nlltape*  
    (*formula-n*  $\Phi_0$  @ *formula-n*  $\Phi_1$  @ *formula-n*  $\Phi_2$  @ *formula-n*  $\Phi_3$  @ *formula-n*  $\Phi_4$  @  
    *formula-n*  $\Phi_5$  @ *formula-n*  $\Phi_6$  @ *formula-n*  $\Phi_7$  @ *formula-n*  $\Phi_8$ ),  
91 := ( $\lfloor N \rfloor_N, 1$ ),  
92 := ( $\lfloor H \rfloor_N, 1$ ),  
93 := ( $\lfloor Z \rfloor_N, 1$ ),  
94 := ( $\lfloor T' \rfloor_N, 1$ ),  
95 := ( $\lfloor \text{formula-n } \psi \rfloor_{NLL}, 1$ ),  
96 := ( $\lfloor \text{formula-n } \psi' \rfloor_{NLL}, 1$ )

**lemma** *tm53* [*transforms-intros*]:

**assumes**  $ttt = ttt47 + 68 + 3 * \text{nlength } N + 2 * \text{nlength } H + 2 * \text{nlength } Z + 3 * \text{nlength } T' +$   
     $2 * \text{nlllength } (\text{formula-n } \psi) + 2 * \text{length } (\text{numlistlist } (\text{formula-n } \psi'))$   
**shows** *transforms tm53 tps0 ttt tps53*  
*<proof>*

**definition** *tps54*  $\equiv$  *tpsJ*

[11 := ( $\lfloor n \rfloor_N, 1$ ),  
15 := ( $\lfloor p \ n \rfloor_N, 1$ ),  
16 := ( $\lfloor m \rfloor_N, 1$ ),  
17 := ( $\lfloor T' \rfloor_N, 1$ ),  
4 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 0) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),  
7 := ( $\lfloor \text{map } (\lambda t. \text{exc } zs \ t \ <\#\> \ 1) \ [0..\<Suc \ T'] \rfloor_{NL}, 1$ ),  
18 := ( $\lfloor m' \rfloor_N, 1$ ),  
19 := ( $\lfloor H \rfloor_N, 1$ ),  
20 := ( $\lfloor m' * H \rfloor_N, 1$ ),

$1 := \text{nlldatape}$   
 $(\text{formula-n } \Phi_0 @ \text{formula-n } \Phi_1 @ \text{formula-n } \Phi_2 @ \text{formula-n } \Phi_3 @ \text{formula-n } \Phi_4 @$   
 $\text{formula-n } \Phi_5 @ \text{formula-n } \Phi_6 @ \text{formula-n } \Phi_7 @ \text{formula-n } \Phi_8),$   
 $91 := (\lfloor N \rfloor_N, 1),$   
 $92 := (\lfloor H \rfloor_N, 1),$   
 $93 := (\lfloor Z \rfloor_N, 1),$   
 $94 := (\lfloor T' \rfloor_N, 1),$   
 $95 := (\lfloor \text{formula-n } \psi \rfloor_{NLL}, 1),$   
 $96 := (\lfloor \text{formula-n } \psi' \rfloor_{NLL}, 1),$   
 $97 := (\lfloor 1 \rfloor_N, 1)$

**lemma**  $tm54$  [*transforms-intros*]:

**assumes**  $t_{tt} = t_{tt}47 + 80 + 3 * \text{nlength } N + 2 * \text{nlength } H + 2 * \text{nlength } Z + 3 * \text{nlength } T' +$   
 $2 * \text{nllength } (\text{formula-n } \psi) + 2 * \text{nllength } (\text{formula-n } \psi')$

**shows**  $\text{transforms } tm54 \text{ tps0 } t_{tt} \text{ tps54}$

*<proof>*

**definition**  $tps55 \equiv tpsJ$

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\#> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlldatape } (\text{formula-n } PHI),$   
 $91 := (\lfloor N \rfloor_N, 1),$   
 $92 := (\lfloor H \rfloor_N, 1),$   
 $93 := (\lfloor Z \rfloor_N, 1),$   
 $94 := (\lfloor T' \rfloor_N, 1),$   
 $95 := (\lfloor \text{formula-n } \psi \rfloor_{NLL}, 1),$   
 $96 := (\lfloor \text{formula-n } \psi' \rfloor_{NLL}, 1),$   
 $97 := (\lfloor 1 \rfloor_N, 1),$   
 $91 + 6 := (\lfloor Suc T' \rfloor_N, 1),$   
 $91 + 3 := (\lfloor 0 \rfloor_N, 1)$

**definition**  $t_{tt}55 \equiv t_{tt}47 + 80 + 3 * \text{nlength } N + 2 * \text{nlength } H + 2 * \text{nlength } Z +$   
 $3 * \text{nlength } T' + 2 * \text{nllength } (\text{formula-n } \psi) + 2 * \text{nllength } (\text{formula-n } \psi') +$   
 $16114767 * 2 ^ (16 * Z) * N ^ 7$

**lemma**  $t_{tt}55$ :  $t_{tt}55 \leq t_{tt}47 + 2 * \text{nllength } (\text{formula-n } \psi) + 2 * \text{nllength } (\text{formula-n } \psi') +$   
 $16114857 * 2 ^ (16 * Z) * N ^ 7$

*<proof>*

**lemma**  $tm55$  [*transforms-intros*]:  $\text{transforms } tm55 \text{ tps0 } t_{tt}55 \text{ tps55}$

*<proof>*

**lemma**  $tps0\text{-start-config}$ :  $(0, tps0) = \text{start-config } 110 \text{ xs}$

*<proof>*

**lemma**  $tm55'$ :  $\text{snd } (\text{execute } tm55 \text{ (start-config } 110 \text{ xs) } t_{tt}55) = tps55$

*<proof>*

**definition**  $tpsK \equiv tpsJ$

$[91 := (\lfloor N \rfloor_N, 1),$   
 $92 := (\lfloor H \rfloor_N, 1),$   
 $93 := (\lfloor Z \rfloor_N, 1),$   
 $94 := (\lfloor T' \rfloor_N, 1),$   
 $95 := (\lfloor \text{formula-n } \psi \rfloor_{NLL}, 1),$   
 $96 := (\lfloor \text{formula-n } \psi' \rfloor_{NLL}, 1),$   
 $97 := (\lfloor 1 \rfloor_N, 1),$

$91 + 6 := (\lfloor \text{Suc } T' \rfloor_N, 1),$   
 $91 + 3 := (\lfloor 0 \rfloor_N, 1)$

**lemma** *tpsK*:  $97 < j \implies j < 110 \implies \text{tpsK } ! j = (\lfloor \square \rfloor, 1)$   
 ⟨proof⟩

**lemma** *lentpsK*:  $\text{length } \text{tpsK} = 110$   
 ⟨proof⟩

**lemma** *tps55*:  $\text{tps55} = \text{tpsK}$

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } (\text{formula-n } \text{PHI})$   
 ⟨proof⟩

**definition** *tps56*  $\equiv \text{tpsK}$

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := (\lfloor \text{formula-n } \text{PHI} \rfloor_{NLL}, 1)$

**lemma** *tm56* [*transforms-intros*]:  
**assumes**  $\text{tnt} = \text{tnt55} + \text{tps55} : \# : 1 + 2$   
**shows** *transforms tm56 tps0 tnt tps56*  
 ⟨proof⟩

**definition** *tps57*  $\equiv \text{tpsK}$

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := \text{nlltape } (\text{formula-n } \text{PHI}),$   
 $109 := \text{nlltape } (\text{formula-n } \text{PHI})$

**lemma** *tm57* [*transforms-intros*]:  
**assumes**  $\text{tnt} = \text{tnt55} + \text{tps55} : \# : 1 + 2 + \text{Suc } (\text{nlllength } (\text{formula-n } \text{PHI}))$   
**shows** *transforms tm57 tps0 tnt tps57*  
 ⟨proof⟩

**definition** *tps58*  $\equiv \text{tpsK}$

$11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p \ n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<\text{Suc } T'] \rfloor_{NL}, 1),$

$7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := (\lfloor [] \rfloor, 1),$   
 $109 := \text{nulltape (formula-n PHI)}$

**lemma** *tm58* [*transforms-intros*]:

**assumes**  $ttt = ttt55 + 9 + tps55 \text{ :\#} : 1 + 3 * \text{nllength (formula-n PHI)} + tps57 \text{ :\#} : 1$   
**shows** *transforms tm58 tps0 ttt tps58*  
*<proof>*

**definition** *tps59*  $\equiv$  *tpsK*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := (\lfloor [] \rfloor, 1),$   
 $109 := (\lfloor \text{formula-n PHI} \rfloor_{NLL}, 1)]$

**lemma** *tm59* [*transforms-intros*]:

**assumes**  $ttt = ttt55 + 11 + tps55 \text{ :\#} : 1 + 3 * \text{nllength (formula-n PHI)} + tps57 \text{ :\#} : 1 + tps58 \text{ :\#} : 109$   
**shows** *transforms tm59 tps0 ttt tps59*  
*<proof>*

**definition** *tps60*  $\equiv$  *tpsK*

$[11 := (\lfloor n \rfloor_N, 1),$   
 $15 := (\lfloor p n \rfloor_N, 1),$   
 $16 := (\lfloor m \rfloor_N, 1),$   
 $17 := (\lfloor T' \rfloor_N, 1),$   
 $4 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 0) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $7 := (\lfloor \text{map } (\lambda t. \text{exc } z s t <\#\> 1) [0..<Suc T'] \rfloor_{NL}, 1),$   
 $18 := (\lfloor m' \rfloor_N, 1),$   
 $19 := (\lfloor H \rfloor_N, 1),$   
 $20 := (\lfloor m' * H \rfloor_N, 1),$   
 $1 := (\lfloor [] \rfloor, 1),$   
 $109 := (\lfloor \text{formula-n PHI} \rfloor_{NLL}, 1),$   
 $109 := (\lfloor \text{numlistlist (formula-n PHI)} \rfloor,$   
 $\quad \text{Suc (length (numlistlist (formula-n PHI))))},$   
 $1 := (\lfloor \text{binencode (numlistlist (formula-n PHI))} \rfloor,$   
 $\quad \text{Suc (2 * length (numlistlist (formula-n PHI))))}]$

**lemma** *tm60*:

**assumes**  $ttt = ttt55 + 12 + tps55 \text{ :\#} : 1 + 12 * \text{nllength (formula-n PHI)} + tps57 \text{ :\#} : 1 + tps58 \text{ :\#} : 109$   
**shows** *transforms tm60 tps0 ttt tps60*  
*<proof>*

**definition** *ttt60*  $\equiv$   $16 * ttt55$

**lemma** *tm60'*: *transforms tm60 tps0 ttt60 tps60*

*<proof>*

**lemma** *tm60-start-config*: *transforms tm60 (snd (start-config 110 (string-to-symbols x))) ttt60 tps60*

*<proof>*

**end**

**end**

The time bound  $ttt60$  formally depends on the string  $x$ . But we need a bound depending only on the length.

**context** *reduction-sat*  
**begin**

**definition**  $T60 :: nat \Rightarrow nat$  **where**  
 $T60\ nn \equiv reduction\text{-}sat\text{-}x.ttt60\ M\ G\ p\ (replicate\ nn\ True)$

**lemma**  $T60$ :  
**fixes**  $x :: string$   
**shows**  $T60\ (length\ x) = reduction\text{-}sat\text{-}x.ttt60\ M\ G\ p\ x$   
 $\langle proof \rangle$

**lemma** *poly-T60: big-oh-poly*  $T60$   
 $\langle proof \rangle$

This is the function, in terms of bit strings, that maps  $x$  to  $\Phi$ .

**definition**  $f_{reduce} :: string \Rightarrow string$  ( $\langle f_{reduce} \rangle$ ) **where**  
 $f_{reduce}\ x \equiv formula\text{-}to\text{-}string\ (reduction\text{-}sat\text{-}x.PHI\ M\ G\ p\ x)$

The function  $f_{reduce}$  many-one reduces  $L$  to SAT.

**lemma** *x-in-L*:  $x \in L \iff f_{reduce}\ x \in SAT$   
 $\langle proof \rangle$

The Turing machine  $tm60$  computes  $f_{reduce}$  with time bound  $T60$ .

**lemma** *computes-in-time-tm60*: *computes-in-time*  $110\ tm60\ f_{reduce}\ T60$   
 $\langle proof \rangle$

Since  $T60$  is bounded by a polynomial, the previous three lemmas imply that  $L$  is polynomial-time many-one reducible to SAT.

**lemma** *L-reducible-SAT*:  $L \leq_p SAT$   
 $\langle proof \rangle$

**end**

In the locale *reduction-sat* the language  $L$  was chosen arbitrarily with properties that we have proven  $\mathcal{NP}$  languages have. So we can now show that SAT is  $\mathcal{NP}$ -hard.

**theorem** *NP-hard-SAT*:  
**assumes**  $L \in \mathcal{NP}$   
**shows**  $L \leq_p SAT$   
 $\langle proof \rangle$

## 8.4 SAT is $\mathcal{NP}$ -complete

The time has come to reap the fruits of our labor and show that SAT is  $\mathcal{NP}$ -complete.

**theorem** *NP-complete-SAT*: *NP-complete*  $SAT$   
 $\langle proof \rangle$

**end**

# Bibliography

- [1] Scott Aaronson. Complexity zoo. [https://complexityzoo.net/Complexity\\_Zoo](https://complexityzoo.net/Complexity_Zoo).
- [2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2006.
- [3] Robert S. Boyer, J Strother Moore, and Matt Kaufmann. ACL2. <https://www.cs.utexas.edu/users/moore/acl2/>.
- [4] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.
- [5] Christian Dalvit and René Thiemann. A verified translation of multitape turing machines into singletape turing machines. *Archive of Formal Proofs*, November 2022. [https://isa-afp.org/entries/Multitape\\_To\\_Singletape\\_TM.html](https://isa-afp.org/entries/Multitape_To_Singletape_TM.html), Formal proof development.
- [6] Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *ITP*, 2017.
- [7] Lennard Gäher and Fabian Kunze. Mechanising complexity theory: The Cook-Levin theorem in Coq. In *ITP*, 2021.
- [8] Ruben Gamboa and John R. Cowles. A mechanical proof of the Cook-Levin theorem. In *TPHOLs*, 2004.
- [9] Juris Hartmanis and Richard Edwin Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.
- [10] L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.
- [11] Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Texts in Computer Science. Springer, 4th edition, 2019.
- [12] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2nd edition, 2006.
- [13] The Coq Development Team. The Coq proof assistant, January 2022.
- [14] René Thiemann and Lukas Schmidinger. The generalized multiset ordering is NP-complete. *Archive of Formal Proofs*, April 2022. [https://isa-afp.org/entries/Multiset\\_Ordering\\_NPC.html](https://isa-afp.org/entries/Multiset_Ordering_NPC.html), Formal proof development.
- [15] Jian Xu, Xingyuan Zhang, Christian Urban, and Sebastiaan J. C. Joosten. Universal turing machine. *Archive of Formal Proofs*, February 2019. [http://isa-afp.org/entries/Universal\\_Turing\\_Machine.html](http://isa-afp.org/entries/Universal_Turing_Machine.html), Formal proof development.