

Some classical results in inductive inference of recursive
functions

Frank J. Balbach

February 6, 2026

Abstract

This entry formalizes some classical concepts and results from inductive inference of recursive functions. In the basic setting a partial recursive function (“strategy”) must identify (“learn”) all functions from a set (“class”) of recursive functions. To that end the strategy receives more and more values $f(0), f(1), f(2), \dots$ of some function f from the given class and in turn outputs descriptions of partial recursive functions, for example, Gödel numbers. The strategy is considered successful if the sequence of outputs (“hypotheses”) converges to a description of f . A class of functions learnable in this sense is called “learnable in the limit”. The set of all these classes is denoted by LIM.

Other types of inference considered are finite learning (FIN), behaviorally correct learning in the limit (BC), and some variants of LIM with restrictions on the hypotheses: total learning (TOTAL), consistent learning (CONS), and class-preserving learning (CP). The main results formalized are the proper inclusions $\text{FIN} \subset \text{CP} \subset \text{TOTAL} \subset \text{CONS} \subset \text{LIM} \subset \text{BC} \subset 2^{\mathcal{R}}$, where \mathcal{R} is the set of all total recursive functions. Further results show that for all these inference types except CONS, strategies can be assumed to be total recursive functions; that all inference types but CP are closed under the subset relation between classes; and that no inference type is closed under the union of classes.

The above is based on a formalization of recursive functions heavily inspired by the *Universal Turing Machine* entry by Xu et al. [18], but different in that it models partial functions with codomain *nat option*. The formalization contains a construction of a universal partial recursive function, without resorting to Turing machines, introduces decidability and recursive enumerability, and proves some standard results: existence of a Kleene normal form, the *s-m-n* theorem, Rice’s theorem, and assorted fixed-point theorems (recursion theorems) by Kleene, Rogers, and Smullyan.

Contents

1	Partial recursive functions	3
1.1	Basic definitions	3
1.1.1	Partial recursive functions	3
1.1.2	Extensional equality	8
1.1.3	Primitive recursive and total functions	10
1.2	Simple functions	11
1.2.1	Manipulating parameters	12
1.2.2	Arithmetic and logic	13
1.2.3	Comparison and conditions	15
1.3	The halting problem	17
1.4	Encoding tuples and lists	17
1.4.1	Pairs and tuples	18
1.4.2	Lists	25
1.5	A universal partial recursive function	36
1.5.1	A step function	36
1.5.2	Encoding partial recursive functions	50
1.5.3	The step function on encoded configurations	53
1.5.4	The step function as a partial recursive function	60
1.5.5	The universal function	64
1.6	Applications of the universal function	69
1.6.1	Lazy conditional evaluation	70
1.6.2	Enumerating the domains of partial recursive functions	70
1.6.3	Concurrent evaluation of functions	76
1.7	Kleene normal form and the number of μ -operations	81
1.8	The s - m - n theorem	84
1.9	Fixed-point theorems	91
1.9.1	Rogers's fixed-point theorem	91
1.9.2	Kleene's fixed-point theorem	93
1.9.3	Smullyan's double fixed-point theorem	95
1.10	Decidable and recursively enumerable sets	95
1.11	Rice's theorem	100
1.12	Partial recursive functions as actual functions	101
1.12.1	The definitions	101
1.12.2	Some simple properties	104
1.12.3	The Gödel numbering φ	106
1.12.4	Fixed-point theorems	108

2	Inductive inference of recursive functions	109
2.1	Preliminaries	110
2.1.1	The prefixes of a function	110
2.1.2	NUM	114
2.2	Types of inference	120
2.2.1	LIM: Learning in the limit	120
2.2.2	BC: Behaviorally correct learning in the limit	122
2.2.3	CONS: Learning in the limit with consistent hypotheses	124
2.2.4	TOTAL: Learning in the limit with total hypotheses	126
2.2.5	CP: Learning in the limit with class-preserving hypotheses	127
2.2.6	FIN: Finite learning	128
2.3	FIN is a proper subset of CP	129
2.4	NUM and FIN are incomparable	137
2.5	NUM and CP are incomparable	140
2.6	NUM is a proper subset of TOTAL	142
2.7	CONS is a proper subset of LIM	147
2.8	Lemma R	157
2.8.1	Strong Lemma R for LIM, FIN, and BC	157
2.8.2	Weaker Lemma R for CP and TOTAL	167
2.8.3	No Lemma R for CONS	168
2.9	LIM is a proper subset of BC	193
2.9.1	Enumerating enough total strategies	194
2.9.2	The diagonalization process	195
2.9.3	The separating class	211
2.9.4	The separating class is in BC	214
2.10	TOTAL is a proper subset of CONS	216
2.10.1	TOTAL is a subset of CONS	216
2.10.2	The separating class	219
2.11	\mathcal{R} is not in BC	241
2.12	The union of classes	250

Chapter 1

Partial recursive functions

```
theory Partial-Recursive
  imports Main HOL-Library.Nat-Bijection
begin
```

This chapter lays the foundation for Chapter 2. Essentially it develops recursion theory up to the point of certain fixed-point theorems. This in turn requires standard results such as the existence of a universal function and the *s-m-n* theorem. Besides these, the chapter contains some results, mostly regarding decidability and the Kleene normal form, that are not strictly needed later. They are included as relatively low-hanging fruits to round off the chapter.

The formalization of partial recursive functions is very much inspired by the Universal Turing Machine AFP entry by Xu et al. [18]. It models partial recursive functions as algorithms whose semantics is given by an evaluation function. This works well for most of this chapter. For the next chapter, however, it is beneficial to regard partial recursive functions as “proper” partial functions. To that end, Section 1.12 introduces more conventional and convenient notation for the common special cases of unary and binary partial recursive functions.

Especially for the nontrivial proofs I consulted the classical textbook by Rogers [12], which also partially explains my preferring the traditional term “recursive” to the more modern “computable”.

1.1 Basic definitions

1.1.1 Partial recursive functions

To represent partial recursive functions we start from the same datatype as Xu et al. [18], more specifically from Urban’s version of the formalization. In fact the datatype *recf* and the function *arity* below have been copied verbatim from it.

```
datatype recf =
  Z
| S
| Id nat nat
| Cn nat recf recf list
| Pr nat recf recf
| Mn nat recf
```

```
fun arity :: recf  $\Rightarrow$  nat where
```

```

    arity Z = 1
|  arity S = 1
|  arity (Id m n) = m
|  arity (Cn n f gs) = n
|  arity (Pr n f g) = Suc n
|  arity (Mn n f) = n

```

Already we deviate from Xu et al. in that we define a well-formedness predicate for partial recursive functions. Well-formedness essentially means arity constraints are respected when combining *recfs*.

```

fun wellf :: recf ⇒ bool where
  wellf Z = True
|  wellf S = True
|  wellf (Id m n) = (n < m)
|  wellf (Cn n f gs) =
    (n > 0 ∧ (∀ g ∈ set gs. arity g = n ∧ wellf g) ∧ arity f = length gs ∧ wellf f)
|  wellf (Pr n f g) =
    (arity g = Suc (Suc n) ∧ arity f = n ∧ wellf f ∧ wellf g)
|  wellf (Mn n f) = (n > 0 ∧ arity f = Suc n ∧ wellf f)

```

lemma *wellf-arity-nonzero*: $\text{wellf } f \implies \text{arity } f > 0$
by (*induction f rule: arity.induct*) *simp-all*

lemma *wellf-Pr-arity-greater-1*: $\text{wellf } (\text{Pr } n \ f \ g) \implies \text{arity } (\text{Pr } n \ f \ g) > 1$
using *wellf-arity-nonzero* **by** *auto*

For the most part of this chapter this is the meaning of “*f* is an *n*-ary partial recursive function”:

abbreviation *recfn* :: $\text{nat} \Rightarrow \text{recf} \Rightarrow \text{bool}$ **where**
recfn *n f* ≡ $\text{wellf } f \wedge \text{arity } f = n$

Some abbreviations for working with *nat option*:

abbreviation *divergent* :: $\text{nat option} \Rightarrow \text{bool}$ ($\langle \cdot \uparrow \rangle [50] 50$) **where**
 $x \uparrow \equiv x = \text{None}$

abbreviation *convergent* :: $\text{nat option} \Rightarrow \text{bool}$ ($\langle \cdot \downarrow \rangle [50] 50$) **where**
 $x \downarrow \equiv x \neq \text{None}$

abbreviation *convergent-eq* :: $\text{nat option} \Rightarrow \text{nat} \Rightarrow \text{bool}$ (**infix** $\langle \downarrow = \rangle 50$) **where**
 $x \downarrow = y \equiv x = \text{Some } y$

abbreviation *convergent-neq* :: $\text{nat option} \Rightarrow \text{nat} \Rightarrow \text{bool}$ (**infix** $\langle \downarrow \neq \rangle 50$) **where**
 $x \downarrow \neq y \equiv x \downarrow \wedge x \neq \text{Some } y$

In prose the terms “halt”, “terminate”, “converge”, and “defined” will be used interchangeably; likewise for “not halt”, “diverge”, and “undefined”. In names of lemmas, the abbreviations *converg* and *diverg* will be used consistently.

Our second major deviation from Xu et al. [18] is that we model the semantics of a *recf* by combining the value and the termination of a function into one evaluation function with codomain *nat option*, rather than separating both aspects into an evaluation function with codomain *nat* and a termination predicate.

The value of a well-formed partial recursive function applied to a correctly-sized list of arguments:

```

fun eval-welf :: recf ⇒ nat list ⇒ nat option where
  eval-welf Z xs ↓= 0
| eval-welf S xs ↓= Suc (xs ! 0)
| eval-welf (Id m n) xs ↓= xs ! n
| eval-welf (Cn n f gs) xs =
  (if ∀ g ∈ set gs. eval-welf g xs ↓
   then eval-welf f (map (λg. the (eval-welf g xs)) gs)
   else None)
| eval-welf (Pr n f g) [] = undefined
| eval-welf (Pr n f g) (0 # xs) = eval-welf f xs
| eval-welf (Pr n f g) (Suc x # xs) =
  Option.bind (eval-welf (Pr n f g) (x # xs)) (λv. eval-welf g (x # v # xs))
| eval-welf (Mn n f) xs =
  (let E = λz. eval-welf f (z # xs)
   in if ∃ z. E z ↓= 0 ∧ (∀ y < z. E y ↓)
   then Some (LEAST z. E z ↓= 0 ∧ (∀ y < z. E y ↓))
   else None)

```

We define a function value only if the *recf* is well-formed and its arity matches the number of arguments.

```

definition eval :: recf ⇒ nat list ⇒ nat option where
  recfn (length xs) f ⇒ eval f xs ≡ eval-welf f xs

```

```

lemma eval-Z [simp]: eval Z [x] ↓= 0
by (simp add: eval-def)

```

```

lemma eval-Z' [simp]: length xs = 1 ⇒ eval Z xs ↓= 0
by (simp add: eval-def)

```

```

lemma eval-S [simp]: eval S [x] ↓= Suc x
by (simp add: eval-def)

```

```

lemma eval-S' [simp]: length xs = 1 ⇒ eval S xs ↓= Suc (hd xs)
using eval-def hd-conv-nth[of xs] by fastforce

```

```

lemma eval-Id [simp]:
assumes n < m and m = length xs
shows eval (Id m n) xs ↓= xs ! n
using assms by (simp add: eval-def)

```

```

lemma eval-Cn [simp]:
assumes recfn (length xs) (Cn n f gs)
shows eval (Cn n f gs) xs =
  (if ∀ g ∈ set gs. eval g xs ↓
   then eval f (map (λg. the (eval g xs)) gs)
   else None)

```

proof –

```

have eval (Cn n f gs) xs = eval-welf (Cn n f gs) xs
using assms eval-def by blast
moreover have ∧g. g ∈ set gs ⇒ eval-welf g xs = eval g xs
using assms eval-def by simp
ultimately have eval (Cn n f gs) xs =
  (if ∀ g ∈ set gs. eval g xs ↓
   then eval-welf f (map (λg. the (eval g xs)) gs)
   else None)
using map-eq-conv[of λg. the (eval-welf g xs) gs λg. the (eval g xs)]

```

by (*auto, metis*)
moreover have $\bigwedge ys. \text{length } ys = \text{length } gs \implies \text{eval } f \text{ } ys = \text{eval-wellf } f \text{ } ys$
 using *assms eval-def* **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

lemma *eval-Pr-0* [*simp*]:
assumes *recfn* (*Suc n*) (*Pr n f g*) **and** $n = \text{length } xs$
shows $\text{eval } (Pr \ n \ f \ g) \ (0 \ \# \ xs) = \text{eval } f \ xs$
using *assms* **by** (*simp add: eval-def*)

lemma *eval-Pr-diverg-Suc* [*simp*]:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and $n = \text{length } xs$
and $\text{eval } (Pr \ n \ f \ g) \ (x \ \# \ xs) \uparrow$
shows $\text{eval } (Pr \ n \ f \ g) \ (Suc \ x \ \# \ xs) \uparrow$
using *assms* **by** (*simp add: eval-def*)

lemma *eval-Pr-converg-Suc* [*simp*]:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and $n = \text{length } xs$
and $\text{eval } (Pr \ n \ f \ g) \ (x \ \# \ xs) \downarrow$
shows $\text{eval } (Pr \ n \ f \ g) \ (Suc \ x \ \# \ xs) = \text{eval } g \ (x \ \# \ \text{the } (\text{eval } (Pr \ n \ f \ g) \ (x \ \# \ xs)) \ \# \ xs)$
using *assms eval-def* **by** *auto*

lemma *eval-Pr-diverg-add*:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and $n = \text{length } xs$
and $\text{eval } (Pr \ n \ f \ g) \ (x \ \# \ xs) \uparrow$
shows $\text{eval } (Pr \ n \ f \ g) \ ((x + y) \ \# \ xs) \uparrow$
using *assms* **by** (*induction y*) *auto*

lemma *eval-Pr-converg-le*:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and $n = \text{length } xs$
and $\text{eval } (Pr \ n \ f \ g) \ (x \ \# \ xs) \downarrow$
and $y \leq x$
shows $\text{eval } (Pr \ n \ f \ g) \ (y \ \# \ xs) \downarrow$
using *assms eval-Pr-diverg-add le-Suc-ex* **by** *metis*

lemma *eval-Pr-Suc-converg*:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and $n = \text{length } xs$
and $\text{eval } (Pr \ n \ f \ g) \ (Suc \ x \ \# \ xs) \downarrow$
shows $\text{eval } g \ (x \ \# \ (\text{the } (\text{eval } (Pr \ n \ f \ g) \ (x \ \# \ xs)))) \ \# \ xs) \downarrow$
and $\text{eval } (Pr \ n \ f \ g) \ (Suc \ x \ \# \ xs) = \text{eval } g \ (x \ \# \ \text{the } (\text{eval } (Pr \ n \ f \ g) \ (x \ \# \ xs)) \ \# \ xs)$
using *eval-Pr-converg-Suc*[*of n f g xs x, OF assms(1,2)*]
eval-Pr-converg-le[*of n f g xs Suc x x, OF assms*] *assms(3)*
by *simp-all*

lemma *eval-Mn* [*simp*]:
assumes *recfn* (*length xs*) (*Mn n f*)
shows $\text{eval } (Mn \ n \ f) \ xs =$
(if $(\exists z. \text{eval } f \ (z \ \# \ xs) \downarrow = 0 \wedge (\forall y < z. \text{eval } f \ (y \ \# \ xs) \downarrow))$
then $\text{Some } (\text{LEAST } z. \text{eval } f \ (z \ \# \ xs) \downarrow = 0 \wedge (\forall y < z. \text{eval } f \ (y \ \# \ xs) \downarrow))$
else *None*)

using *assms eval-def* by *auto*

For μ -recursion, the condition $\forall y < z. \text{eval-wellf } f (y \# xs) \downarrow$ inside *LEAST* in the definition of *eval-wellf* is redundant.

lemma *eval-wellf-Mn*:

eval-wellf (Mn n f) xs =
 (if ($\exists z. \text{eval-wellf } f (z \# xs) \downarrow = 0 \wedge (\forall y < z. \text{eval-wellf } f (y \# xs) \downarrow$))
 then *Some* (*LEAST* z. *eval-wellf* f (z # xs) $\downarrow = 0$)
 else *None*)

proof –

let $?P = \lambda z. \text{eval-wellf } f (z \# xs) \downarrow = 0 \wedge (\forall y < z. \text{eval-wellf } f (y \# xs) \downarrow$
 {
 assume $\exists z. ?P z$
 moreover define z where z = *Least* ?P
 ultimately have ?P z
 using *LeastI*[of ?P] by *blast*
 have (*LEAST* z. *eval-wellf* f (z # xs) $\downarrow = 0$) = z
 proof (rule *Least-equality*)
 show *eval-wellf* f (z # xs) $\downarrow = 0$
 using $\langle ?P z \rangle$ by *simp*
 show $z \leq y$ if *eval-wellf* f (y # xs) $\downarrow = 0$ for y
 proof (rule *ccontr*)
 assume $\neg z \leq y$
 then have $y < z$ by *simp*
 moreover from this have ?P y
 using that $\langle ?P z \rangle$ by *simp*
 ultimately show *False*
 using that *not-less-Least z-def* by *blast*
 qed
 qed
 }
 then show ?thesis by *simp*

qed

lemma *eval-Mn'*:

assumes *recfn* (length xs) (Mn n f)
 shows *eval* (Mn n f) xs =
 (if ($\exists z. \text{eval } f (z \# xs) \downarrow = 0 \wedge (\forall y < z. \text{eval } f (y \# xs) \downarrow$))
 then *Some* (*LEAST* z. *eval* f (z # xs) $\downarrow = 0$)
 else *None*)
 using *assms eval-def eval-wellf-Mn* by *auto*

Proving that μ -recursion converges is easier if one does not have to deal with *LEAST* directly.

lemma *eval-Mn-convergI*:

assumes *recfn* (length xs) (Mn n f)
 and *eval* f (z # xs) $\downarrow = 0$
 and $\bigwedge y. y < z \implies \text{eval } f (y \# xs) \downarrow \neq 0$
 shows *eval* (Mn n f) xs $\downarrow = z$

proof –

let $?P = \lambda z. \text{eval } f (z \# xs) \downarrow = 0 \wedge (\forall y < z. \text{eval } f (y \# xs) \downarrow$
 have z = *Least* ?P
 using *Least-equality*[of ?P z] *assms(2,3)* *not-le-imp-less* by *blast*
 moreover have ?P z using *assms(2,3)* by *simp*
 ultimately show *eval* (Mn n f) xs $\downarrow = z$

using *eval-Mn*[*OF assms*(1)] **by** *meson*
qed

Similarly, reasoning from a μ -recursive function is simplified somewhat by the next lemma.

lemma *eval-Mn-convergE*:

assumes *recfn* (*length xs*) (*Mn n f*) **and** *eval* (*Mn n f*) *xs* $\downarrow = z$
shows $z = (\text{LEAST } z. \text{eval } f (z \# xs) \downarrow = 0 \wedge (\forall y < z. \text{eval } f (y \# xs) \downarrow))$
and *eval* *f* (*z* $\#$ *xs*) $\downarrow = 0$
and $\bigwedge y. y < z \implies \text{eval } f (y \# xs) \downarrow \neq 0$

proof –

let $?P = \lambda z. \text{eval } f (z \# xs) \downarrow = 0 \wedge (\forall y < z. \text{eval } f (y \# xs) \downarrow)$
show $z = \text{Least } ?P$
using *assms eval-Mn*[*OF assms*(1)]
by (*metis* (*no-types*, *lifting*) *option.inject option.simps*(3))
moreover have $\exists z. ?P z$
using *assms eval-Mn*[*OF assms*(1)] **by** (*metis option.distinct*(1))
ultimately have $?P z$
using *LeastI*[*of* $?P$] **by** *blast*
then have *eval* *f* (*z* $\#$ *xs*) $\downarrow = 0 \wedge (\forall y < z. \text{eval } f (y \# xs) \downarrow)$
by *simp*
then show *eval* *f* (*z* $\#$ *xs*) $\downarrow = 0$ **by** *simp*
show $\bigwedge y. y < z \implies \text{eval } f (y \# xs) \downarrow \neq 0$
using *not-less-Least*[*of* - $?P$] $\langle z = \text{Least } ?P \rangle \langle ?P z \rangle$ *less-trans* **by** *blast*

qed

lemma *eval-Mn-diverg*:

assumes *recfn* (*length xs*) (*Mn n f*)
shows $\neg (\exists z. \text{eval } f (z \# xs) \downarrow = 0 \wedge (\forall y < z. \text{eval } f (y \# xs) \downarrow)) \longleftrightarrow \text{eval } (Mn n f) xs \uparrow$
using *assms eval-Mn*[*OF assms*(1)] **by** *simp*

1.1.2 Extensional equality

definition *exteq* :: *recf* \Rightarrow *recf* \Rightarrow *bool* (**infix** $\langle \simeq \rangle$ 55) **where**

$f \simeq g \equiv \text{arity } f = \text{arity } g \wedge (\forall xs. \text{length } xs = \text{arity } f \longrightarrow \text{eval } f xs = \text{eval } g xs)$

lemma *exteq-refl*: $f \simeq f$

using *exteq-def* **by** *simp*

lemma *exteq-sym*: $f \simeq g \implies g \simeq f$

using *exteq-def* **by** *simp*

lemma *exteq-trans*: $f \simeq g \implies g \simeq h \implies f \simeq h$

using *exteq-def* **by** *simp*

lemma *exteqI*:

assumes $\text{arity } f = \text{arity } g$ **and** $\bigwedge xs. \text{length } xs = \text{arity } f \implies \text{eval } f xs = \text{eval } g xs$

shows $f \simeq g$

using *assms exteq-def* **by** *simp*

lemma *exteqI1*:

assumes $\text{arity } f = 1$ **and** $\text{arity } g = 1$ **and** $\bigwedge x. \text{eval } f [x] = \text{eval } g [x]$

shows $f \simeq g$

using *assms exteqI* **by** (*metis* *One-nat-def Suc-length-conv length-0-conv*)

For every partial recursive function f there are infinitely many extensionally equal ones,

for example, those that wrap f arbitrarily often in the identity function.

```
fun wrap-Id :: recf  $\Rightarrow$  nat  $\Rightarrow$  recf where
  wrap-Id f 0 = f
| wrap-Id f (Suc n) = Cn (arity f) (Id 1 0) [wrap-Id f n]
```

```
lemma recfn-wrap-Id: recfn a f  $\Longrightarrow$  recfn a (wrap-Id f n)
using wellf-arity-nonzero by (induction n) auto
```

```
lemma exteq-wrap-Id: recfn a f  $\Longrightarrow$  f  $\simeq$  wrap-Id f n
```

```
proof (induction n)
  case 0
  then show ?case by (simp add: exteq-refl)
next
  case (Suc n)
  have wrap-Id f n  $\simeq$  wrap-Id f (Suc n)
  proof (rule exteqI)
    show arity (wrap-Id f n) = arity (wrap-Id f (Suc n))
      using Suc by (simp add: recfn-wrap-Id)
    show eval (wrap-Id f n) xs = eval (wrap-Id f (Suc n)) xs
      if length xs = arity (wrap-Id f n) for xs
    proof -
      have recfn (length xs) (Cn (arity f) (Id 1 0) [wrap-Id f n])
        using that Suc recfn-wrap-Id by (metis wrap-Id.simps(2))
      then show eval (wrap-Id f n) xs = eval (wrap-Id f (Suc n)) xs
        by auto
    qed
  qed
  then show ?case using Suc exteq-trans by fast
qed
```

```
fun depth :: recf  $\Rightarrow$  nat where
  depth Z = 0
| depth S = 0
| depth (Id m n) = 0
| depth (Cn n f gs) = Suc (max (depth f) (Max (set (map depth gs))))
| depth (Pr n f g) = Suc (max (depth f) (depth g))
| depth (Mn n f) = Suc (depth f)
```

```
lemma depth-wrap-Id: recfn a f  $\Longrightarrow$  depth (wrap-Id f n) = depth f + n
by (induction n) simp-all
```

```
lemma wrap-Id-injective:
  assumes recfn a f and wrap-Id f n1 = wrap-Id f n2
  shows n1 = n2
  using assms by (metis add-left-cancel depth-wrap-Id)
```

```
lemma exteq-infinite:
  assumes recfn a f
  shows infinite {g. recfn a g  $\wedge$  g  $\simeq$  f} (is infinite ?R)
proof -
  have inj (wrap-Id f)
    using wrap-Id-injective  $\langle$ recfn a f $\rangle$  by (meson inj-onI)
  then have infinite (range (wrap-Id f))
    using finite-imageD by blast
  moreover have range (wrap-Id f)  $\subseteq$  ?R
```

using *assms exteq-sym exteq-wrap-Id recfn-wrap-Id* **by** *blast*
ultimately show *?thesis* **by** (*simp add: infinite-super*)
qed

1.1.3 Primitive recursive and total functions

fun *Mn-free* :: *recf* \Rightarrow *bool* **where**
Mn-free *Z* = *True*
| *Mn-free* *S* = *True*
| *Mn-free* (*Id* *m* *n*) = *True*
| *Mn-free* (*Cn* *n* *f* *gs*) = ((\forall *g* \in *set* *gs*. *Mn-free* *g*) \wedge *Mn-free* *f*)
| *Mn-free* (*Pr* *n* *f* *g*) = (*Mn-free* *f* \wedge *Mn-free* *g*)
| *Mn-free* (*Mn* *n* *f*) = *False*

This is our notion of n -ary primitive recursive function:

abbreviation *prim-recfn* :: *nat* \Rightarrow *recf* \Rightarrow *bool* **where**
prim-recfn *n* *f* \equiv *recfn* *n* *f* \wedge *Mn-free* *f*

definition *total* :: *recf* \Rightarrow *bool* **where**
total *f* \equiv \forall *xs*. *length* *xs* = *arity* *f* \longrightarrow *eval* *f* *xs* \downarrow

lemma *totalI* [*intro*]:
assumes \wedge *xs*. *length* *xs* = *arity* *f* \Longrightarrow *eval* *f* *xs* \downarrow
shows *total* *f*
using *assms total-def* **by** *simp*

lemma *totalE* [*simp*]:
assumes *total* *f* **and** *recfn* *n* *f* **and** *length* *xs* = *n*
shows *eval* *f* *xs* \downarrow
using *assms total-def* **by** *simp*

lemma *totalI1* :
assumes *recfn* 1 *f* **and** \wedge *x*. *eval* *f* [*x*] \downarrow
shows *total* *f*
using *assms totalI[of f]* **by** (*metis One-nat-def length-0-conv length-Suc-conv*)

lemma *totalI2*:
assumes *recfn* 2 *f* **and** \wedge *x* *y*. *eval* *f* [*x*, *y*] \downarrow
shows *total* *f*
using *assms totalI[of f]* **by** (*smt length-0-conv length-Suc-conv numeral-2-eq-2*)

lemma *totalI3*:
assumes *recfn* 3 *f* **and** \wedge *x* *y* *z*. *eval* *f* [*x*, *y*, *z*] \downarrow
shows *total* *f*
using *assms totalI[of f]* **by** (*smt length-0-conv length-Suc-conv numeral-3-eq-3*)

lemma *totalI4*:
assumes *recfn* 4 *f* **and** \wedge *w* *x* *y* *z*. *eval* *f* [*w*, *x*, *y*, *z*] \downarrow
shows *total* *f*

proof (*rule totalI[of f]*)
fix *xs* :: *nat* *list*
assume *length* *xs* = *arity* *f*
then have *length* *xs* = *Suc* (*Suc* (*Suc* (*Suc* 0)))
using *assms(1)* **by** *simp*
then obtain *w* *x* *y* *z* **where** *xs* = [*w*, *x*, *y*, *z*]
by (*smt Suc-length-conv length-0-conv*)

then show $eval\ f\ xs \downarrow$ using $assms(2)$ by $simp$
qed

lemma *Mn-free-imp-total* [intro]:

assumes $wellf\ f$ and $Mn\text{-}free\ f$

shows $total\ f$

using $assms$

proof (*induction f rule: Mn-free.induct*)

case $(5\ n\ f\ g)$

have $eval\ (Pr\ n\ f\ g)\ (x\ \#\ xs) \downarrow$ if $length\ (x\ \#\ xs) = arity\ (Pr\ n\ f\ g)$ for $x\ xs$

using 5 that by (*induction x*) $auto$

then show $?case$ by (*metis arity.simps(5) length-Suc-conv totalI*)

qed (*auto simp add: total-def eval-def*)

lemma *prim-recfn-total*: $prim\text{-}recfn\ n\ f \implies total\ f$

using *Mn-free-imp-total* by $simp$

lemma *eval-Pr-prim-Suc*:

assumes $h = Pr\ n\ f\ g$ and $prim\text{-}recfn\ (Suc\ n)\ h$ and $length\ xs = n$

shows $eval\ h\ (Suc\ x\ \#\ xs) = eval\ g\ (x\ \#\ the\ (eval\ h\ (x\ \#\ xs))\ \#\ xs)$

using $assms\ eval\text{-}Pr\text{-}converg\text{-}Suc\ prim\text{-}recfn\text{-}total$ by $simp$

lemma *Cn-total*:

assumes $\forall g \in set\ gs.\ total\ g$ and $total\ f$ and $recfn\ n\ (Cn\ n\ f\ gs)$

shows $total\ (Cn\ n\ f\ gs)$

using $assms$ by (*simp add: totalI*)

lemma *Pr-total*:

assumes $total\ f$ and $total\ g$ and $recfn\ (Suc\ n)\ (Pr\ n\ f\ g)$

shows $total\ (Pr\ n\ f\ g)$

proof –

have $eval\ (Pr\ n\ f\ g)\ (x\ \#\ xs) \downarrow$ if $length\ xs = n$ for $x\ xs$

using *that assms* by (*induction x*) $auto$

then show $?thesis$

using $assms(3)\ totalI$ by (*metis Suc-length-conv arity.simps(5)*)

qed

lemma *eval-Mn-total*:

assumes $recfn\ (length\ xs)\ (Mn\ n\ f)$ and $total\ f$

shows $eval\ (Mn\ n\ f)\ xs =$

(if $(\exists z.\ eval\ f\ (z\ \#\ xs) \downarrow = 0)$

then $Some\ (LEAST\ z.\ eval\ f\ (z\ \#\ xs) \downarrow = 0)$

else $None$)

using $assms$ by $auto$

1.2 Simple functions

This section, too, bears some similarity to Urban’s formalization in Xu et al. [18], but is more minimalistic in scope.

As a general naming rule, instances of *recf* and functions returning such instances get names starting with *r*. Typically, for an *r-xyz* there will be a lemma *r-xyz-recfn* or *r-xyz-prim* establishing its (primitive) recursiveness, arity, and well-formedness. Moreover there will be a lemma *r-xyz* describing its semantics, for which we will sometimes introduce an Isabelle function *xyz*.

1.2.1 Manipulating parameters

Appending dummy parameters:

definition $r\text{-dummy} :: \text{nat} \Rightarrow \text{recf} \Rightarrow \text{recf}$ **where**
 $r\text{-dummy } n f \equiv \text{Cn } (\text{arity } f + n) f (\text{map } (\lambda i. \text{Id } (\text{arity } f + n) i) [0..<\text{arity } f])$

lemma $r\text{-dummy-prim}$ $[simp]$:
 $\text{prim-recfn } a f \Longrightarrow \text{prim-recfn } (a + n) (r\text{-dummy } n f)$
using $\text{wellf-arity-nonzero}$ **by** $(\text{auto simp add: } r\text{-dummy-def})$

lemma $r\text{-dummy-recfn}$ $[simp]$:
 $\text{recfn } a f \Longrightarrow \text{recfn } (a + n) (r\text{-dummy } n f)$
using $\text{wellf-arity-nonzero}$ **by** $(\text{auto simp add: } r\text{-dummy-def})$

lemma $r\text{-dummy}$ $[simp]$:
 $r\text{-dummy } n f = \text{Cn } (\text{arity } f + n) f (\text{map } (\lambda i. \text{Id } (\text{arity } f + n) i) [0..<\text{arity } f])$
unfolding $r\text{-dummy-def}$ **by** simp

lemma $r\text{-dummy-append}$:
assumes $\text{recfn } (\text{length } xs) f$ **and** $\text{length } ys = n$
shows $\text{eval } (r\text{-dummy } n f) (xs @ ys) = \text{eval } f xs$
proof –
let $?r = r\text{-dummy } n f$
let $?gs = \text{map } (\lambda i. \text{Id } (\text{arity } f + n) i) [0..<\text{arity } f]$
have $\text{length } ?gs = \text{arity } f$ **by** simp
moreover have $?gs ! i = (\text{Id } (\text{arity } f + n) i)$ **if** $i < \text{arity } f$ **for** i
by (simp add: that)
moreover have $*$: $\text{eval-wellf } (?gs ! i) (xs @ ys) \downarrow = xs ! i$ **if** $i < \text{arity } f$ **for** i
using that assms **by** $(\text{simp add: nth-append})$
ultimately have $\text{map } (\lambda g. \text{the } (\text{eval-wellf } g (xs @ ys))) ?gs = xs$
by $(\text{metis } (\text{no-types, lifting}) \text{assms}(1) \text{length-map nth-equalityI nth-map option.sel})$
moreover have $\forall g \in \text{set } ?gs. \text{eval-wellf } g (xs @ ys) \downarrow$
using $*$ **by** simp
moreover have $\text{recfn } (\text{length } (xs @ ys)) ?r$
using $\text{assms } r\text{-dummy-recfn}$ **by** fastforce
ultimately show $?thesis$
by $(\text{auto simp add: assms eval-def})$
qed

Shrinking a binary function to a unary one is useful when we want to define a unary function via the Pr operation, which can only construct recfs of arity two or higher.

definition $r\text{-shrink} :: \text{recf} \Rightarrow \text{recf}$ **where**
 $r\text{-shrink } f \equiv \text{Cn } 1 f [\text{Id } 1 0, \text{Id } 1 0]$

lemma $r\text{-shrink-prim}$ $[simp]$: $\text{prim-recfn } 2 f \Longrightarrow \text{prim-recfn } 1 (r\text{-shrink } f)$
by $(\text{simp add: } r\text{-shrink-def})$

lemma $r\text{-shrink-recfn}$ $[simp]$: $\text{recfn } 2 f \Longrightarrow \text{recfn } 1 (r\text{-shrink } f)$
by $(\text{simp add: } r\text{-shrink-def})$

lemma $r\text{-shrink}$ $[simp]$: $\text{recfn } 2 f \Longrightarrow \text{eval } (r\text{-shrink } f) [x] = \text{eval } f [x, x]$
by $(\text{simp add: } r\text{-shrink-def})$

definition $r\text{-swap} :: \text{recf} \Rightarrow \text{recf}$ **where**
 $r\text{-swap } f \equiv \text{Cn } 2 f [\text{Id } 2 1, \text{Id } 2 0]$

lemma *r-swap-recfn* [*simp*]: $\text{recfn } 2 f \implies \text{recfn } 2 (r\text{-swap } f)$
by (*simp add: r-swap-def*)

lemma *r-swap-prim* [*simp*]: $\text{prim-recfn } 2 f \implies \text{prim-recfn } 2 (r\text{-swap } f)$
by (*simp add: r-swap-def*)

lemma *r-swap* [*simp*]: $\text{recfn } 2 f \implies \text{eval } (r\text{-swap } f) [x, y] = \text{eval } f [y, x]$
by (*simp add: r-swap-def*)

Prepending one dummy parameter:

definition *r-shift* :: $\text{recf} \Rightarrow \text{recf}$ **where**
r-shift $f \equiv \text{Cn } (\text{Suc } (\text{arity } f)) f (\text{map } (\lambda i. \text{Id } (\text{Suc } (\text{arity } f)) (\text{Suc } i)) [0..<\text{arity } f])$

lemma *r-shift-prim* [*simp*]: $\text{prim-recfn } a f \implies \text{prim-recfn } (\text{Suc } a) (r\text{-shift } f)$
by (*simp add: r-shift-def*)

lemma *r-shift-recfn* [*simp*]: $\text{recfn } a f \implies \text{recfn } (\text{Suc } a) (r\text{-shift } f)$
by (*simp add: r-shift-def*)

lemma *r-shift* [*simp*]:
assumes $\text{recfn } (\text{length } xs) f$
shows $\text{eval } (r\text{-shift } f) (x \# xs) = \text{eval } f xs$
proof –
let $?r = r\text{-shift } f$
let $?gs = \text{map } (\lambda i. \text{Id } (\text{Suc } (\text{arity } f)) (\text{Suc } i)) [0..<\text{arity } f]$
have $\text{length } ?gs = \text{arity } f$ **by** *simp*
moreover have $?gs ! i = (\text{Id } (\text{Suc } (\text{arity } f)) (\text{Suc } i))$ **if** $i < \text{arity } f$ **for** i
by (*simp add: that*)
moreover have $*$: $\text{eval } (?gs ! i) (x \# xs) \Downarrow = xs ! i$ **if** $i < \text{arity } f$ **for** i
using *assms nth-append that* **by** *simp*
ultimately have $\text{map } (\lambda g. \text{the } (\text{eval } g (x \# xs))) ?gs = xs$
by (*metis (no-types, lifting) assms length-map nth-equalityI nth-map option.sel*)
moreover have $\forall g \in \text{set } ?gs. \text{eval } g (x \# xs) \neq \text{None}$
using $*$ **by** *simp*
ultimately show *?thesis* **using** *r-shift-def assms* **by** *simp*
qed

1.2.2 Arithmetic and logic

The unary constants:

fun *r-const* :: $\text{nat} \Rightarrow \text{recf}$ **where**
r-const $0 = Z$
 $| r\text{-const } (\text{Suc } c) = \text{Cn } 1 S [r\text{-const } c]$

lemma *r-const-prim* [*simp*]: $\text{prim-recfn } 1 (r\text{-const } c)$
by (*induction c*) (*simp-all*)

lemma *r-const* [*simp*]: $\text{eval } (r\text{-const } c) [x] \Downarrow = c$
by (*induction c*) (*simp-all*)

Constants of higher arities:

definition *r-constn* $n c \equiv \text{if } n = 0 \text{ then } r\text{-const } c \text{ else } r\text{-dummy } n (r\text{-const } c)$

lemma *r-constn-prim* [*simp*]: $\text{prim-recfn } (\text{Suc } n) (r\text{-constn } n c)$

unfolding *r-constn-def* **by** *simp*

lemma *r-constn [simp]*: $\text{length } xs = \text{Suc } n \implies \text{eval } (r\text{-constn } n \ c) \ xs \downarrow = c$

unfolding *r-constn-def*

by *simp (metis length-0-conv length-Suc-conv r-const)*

We introduce addition, subtraction, and multiplication, but interestingly enough we can make do without division.

definition *r-add* $\equiv \text{Pr } 1 \ (\text{Id } 1 \ 0) \ (\text{Cn } 3 \ S \ [\text{Id } 3 \ 1])$

lemma *r-add-prim [simp]*: *prim-recfn 2 r-add*

by (*simp add: r-add-def*)

lemma *r-add [simp]*: $\text{eval } r\text{-add } [a, b] \downarrow = a + b$

unfolding *r-add-def* **by** (*induction a*) *simp-all*

definition *r-mul* $\equiv \text{Pr } 1 \ Z \ (\text{Cn } 3 \ r\text{-add } [\text{Id } 3 \ 1, \ \text{Id } 3 \ 2])$

lemma *r-mul-prim [simp]*: *prim-recfn 2 r-mul*

unfolding *r-mul-def* **by** *simp*

lemma *r-mul [simp]*: $\text{eval } r\text{-mul } [a, b] \downarrow = a * b$

unfolding *r-mul-def* **by** (*induction a*) *simp-all*

definition *r-dec* $\equiv \text{Cn } 1 \ (\text{Pr } 1 \ Z \ (\text{Id } 3 \ 0)) \ [\text{Id } 1 \ 0, \ \text{Id } 1 \ 0]$

lemma *r-dec-prim [simp]*: *prim-recfn 1 r-dec*

by (*simp add: r-dec-def*)

lemma *r-dec [simp]*: $\text{eval } r\text{-dec } [a] \downarrow = a - 1$

proof –

have $\text{eval } (\text{Pr } 1 \ Z \ (\text{Id } 3 \ 0)) \ [x, y] \downarrow = x - 1$ **for** $x \ y$

by (*induction x*) *simp-all*

then show *?thesis* **by** (*simp add: r-dec-def*)

qed

definition *r-sub* $\equiv r\text{-swap } (\text{Pr } 1 \ (\text{Id } 1 \ 0) \ (\text{Cn } 3 \ r\text{-dec } [\text{Id } 3 \ 1]))$

lemma *r-sub-prim [simp]*: *prim-recfn 2 r-sub*

unfolding *r-sub-def* **by** *simp*

lemma *r-sub [simp]*: $\text{eval } r\text{-sub } [a, b] \downarrow = a - b$

proof –

have $\text{eval } (\text{Pr } 1 \ (\text{Id } 1 \ 0) \ (\text{Cn } 3 \ r\text{-dec } [\text{Id } 3 \ 1])) \ [x, y] \downarrow = y - x$ **for** $x \ y$

by (*induction x*) *simp-all*

then show *?thesis* **unfolding** *r-sub-def* **by** *simp*

qed

definition *r-sign* $\equiv r\text{-shrink } (\text{Pr } 1 \ Z \ (r\text{-constn } 2 \ 1))$

lemma *r-sign-prim [simp]*: *prim-recfn 1 r-sign*

unfolding *r-sign-def* **by** *simp*

lemma *r-sign [simp]*: $\text{eval } r\text{-sign } [x] \downarrow = (\text{if } x = 0 \ \text{then } 0 \ \text{else } 1)$

proof –

have $\text{eval } (\text{Pr } 1 \ Z \ (r\text{-constn } 2 \ 1)) \ [x, y] \downarrow = (\text{if } x = 0 \ \text{then } 0 \ \text{else } 1)$ **for** $x \ y$

by (induction x) *simp-all*
then show *?thesis unfolding r-sign-def by simp*
qed

In the logical functions, true will be represented by zero, and false will be represented by non-zero as argument and by one as result.

definition $r\text{-not} \equiv Cn\ 1\ r\text{-sub}\ [r\text{-const}\ 1,\ r\text{-sign}]$

lemma $r\text{-not-prim}$ [*simp*]: *prim-recfn 1 r-not*
unfolding $r\text{-not-def}$ **by** *simp*

lemma $r\text{-not}$ [*simp*]: *eval r-not [x] $\Downarrow =$ (if x = 0 then 1 else 0)*
unfolding $r\text{-not-def}$ **by** *simp*

definition $r\text{-nand} \equiv Cn\ 2\ r\text{-not}\ [r\text{-add}]$

lemma $r\text{-nand-prim}$ [*simp*]: *prim-recfn 2 r-nand*
unfolding $r\text{-nand-def}$ **by** *simp*

lemma $r\text{-nand}$ [*simp*]: *eval r-nand [x, y] $\Downarrow =$ (if x = 0 \wedge y = 0 then 1 else 0)*
unfolding $r\text{-nand-def}$ **by** *simp*

definition $r\text{-and} \equiv Cn\ 2\ r\text{-not}\ [r\text{-nand}]$

lemma $r\text{-and-prim}$ [*simp*]: *prim-recfn 2 r-and*
unfolding $r\text{-and-def}$ **by** *simp*

lemma $r\text{-and}$ [*simp*]: *eval r-and [x, y] $\Downarrow =$ (if x = 0 \wedge y = 0 then 0 else 1)*
unfolding $r\text{-and-def}$ **by** *simp*

definition $r\text{-or} \equiv Cn\ 2\ r\text{-sign}\ [r\text{-mul}]$

lemma $r\text{-or-prim}$ [*simp*]: *prim-recfn 2 r-or*
unfolding $r\text{-or-def}$ **by** *simp*

lemma $r\text{-or}$ [*simp*]: *eval r-or [x, y] $\Downarrow =$ (if x = 0 \vee y = 0 then 0 else 1)*
unfolding $r\text{-or-def}$ **by** *simp*

1.2.3 Comparison and conditions

definition $r\text{-ifz} \equiv$
let ifzero = (Cn 3 r-mul [r-dummy 2 r-not, Id 3 1]);
ifnzero = (Cn 3 r-mul [r-dummy 2 r-sign, Id 3 2])
in Cn 3 r-add [ifzero, ifnzero]

lemma $r\text{-ifz-prim}$ [*simp*]: *prim-recfn 3 r-ifz*
unfolding $r\text{-ifz-def}$ **by** *simp*

lemma $r\text{-ifz}$ [*simp*]: *eval r-ifz [cond, val0, val1] $\Downarrow =$ (if cond = 0 then val0 else val1)*
unfolding $r\text{-ifz-def}$ **by** (*simp add: Let-def*)

definition $r\text{-eq} \equiv Cn\ 2\ r\text{-sign}\ [Cn\ 2\ r\text{-add}\ [r\text{-sub},\ r\text{-swap}\ r\text{-sub}]]$

lemma $r\text{-eq-prim}$ [*simp*]: *prim-recfn 2 r-eq*
unfolding $r\text{-eq-def}$ **by** *simp*

lemma *r-eq* [*simp*]: *eval r-eq* [x, y] \Downarrow = (if x = y then 0 else 1)
unfolding *r-eq-def* **by** *simp*

definition *r-ifeq* \equiv Cn 4 *r-ifz* [*r-dummy* 2 *r-eq*, *Id* 4 2, *Id* 4 3]

lemma *r-ifeq-prim* [*simp*]: *prim-recfn* 4 *r-ifeq*
unfolding *r-ifeq-def* **by** *simp*

lemma *r-ifeq* [*simp*]: *eval r-ifeq* [a, b, v₀, v₁] \Downarrow = (if a = b then v₀ else v₁)
unfolding *r-ifeq-def* **using** *r-dummy-append*[of *r-eq* [a, b] [v₀, v₁] 2]
by *simp*

definition *r-neq* \equiv Cn 2 *r-not* [*r-eq*]

lemma *r-neq-prim* [*simp*]: *prim-recfn* 2 *r-neq*
unfolding *r-neq-def* **by** *simp*

lemma *r-neq* [*simp*]: *eval r-neq* [x, y] \Downarrow = (if x = y then 1 else 0)
unfolding *r-neq-def* **by** *simp*

definition *r-ifle* \equiv Cn 4 *r-ifz* [*r-dummy* 2 *r-sub*, *Id* 4 2, *Id* 4 3]

lemma *r-ifle-prim* [*simp*]: *prim-recfn* 4 *r-ifle*
unfolding *r-ifle-def* **by** *simp*

lemma *r-ifle* [*simp*]: *eval r-ifle* [a, b, v₀, v₁] \Downarrow = (if a \leq b then v₀ else v₁)
unfolding *r-ifle-def* **using** *r-dummy-append*[of *r-sub* [a, b] [v₀, v₁] 2]
by *simp*

definition *r-iffless* \equiv Cn 4 *r-ifle* [*Id* 4 1, *Id* 4 0, *Id* 4 3, *Id* 4 2]

lemma *r-iffless-prim* [*simp*]: *prim-recfn* 4 *r-iffless*
unfolding *r-iffless-def* **by** *simp*

lemma *r-iffless* [*simp*]: *eval r-iffless* [a, b, v₀, v₁] \Downarrow = (if a < b then v₀ else v₁)
unfolding *r-iffless-def* **by** *simp*

definition *r-less* \equiv Cn 2 *r-ifle* [*Id* 2 1, *Id* 2 0, *r-constn* 1 1, *r-constn* 1 0]

lemma *r-less-prim* [*simp*]: *prim-recfn* 2 *r-less*
unfolding *r-less-def* **by** *simp*

lemma *r-less* [*simp*]: *eval r-less* [x, y] \Downarrow = (if x < y then 0 else 1)
unfolding *r-less-def* **by** *simp*

definition *r-le* \equiv Cn 2 *r-ifle* [*Id* 2 0, *Id* 2 1, *r-constn* 1 0, *r-constn* 1 1]

lemma *r-le-prim* [*simp*]: *prim-recfn* 2 *r-le*
unfolding *r-le-def* **by** *simp*

lemma *r-le* [*simp*]: *eval r-le* [x, y] \Downarrow = (if x \leq y then 0 else 1)
unfolding *r-le-def* **by** *simp*

Arguments are evaluated eagerly. Therefore *r-ifz*, etc. cannot be combined with a diverging function to implement a conditionally diverging function in the naive way. The following function implements a special case needed in the next section. A [general](#)

lazy version of *r-ifz* will be introduced later with the help of a universal function.

definition *r-ifeq-else-diverg* \equiv

Cn 3 r-add [Id 3 2, Mn 3 (Cn 4 r-add [Id 4 0, Cn 4 r-eq [Id 4 1, Id 4 2]])]

lemma *r-ifeq-else-diverg-recfn* [*simp*]: *recfn 3 r-ifeq-else-diverg*

unfolding *r-ifeq-else-diverg-def* **by** *simp*

lemma *r-ifeq-else-diverg* [*simp*]:

eval r-ifeq-else-diverg [a, b, v] = (if a = b then Some v else None)

unfolding *r-ifeq-else-diverg-def* **by** *simp*

1.3 The halting problem

Decidability will be treated more thoroughly in Section 1.10. But the halting problem is prominent enough to deserve an early mention.

definition *decidable* :: *nat set* \Rightarrow *bool* **where**

decidable X $\equiv \exists f. \text{recfn } 1 f \wedge (\forall x. \text{eval } f [x] \downarrow = (\text{if } x \in X \text{ then } 1 \text{ else } 0))$

No matter how partial recursive functions are encoded as natural numbers, the set of all codes of functions halting on their own code is undecidable.

theorem *halting-problem-undecidable*:

fixes *code* :: *nat* \Rightarrow *recf*

assumes $\bigwedge f. \text{recfn } 1 f \implies \exists i. \text{code } i = f$

shows $\neg \text{decidable } \{x. \text{eval } (\text{code } x) [x] \downarrow\}$ (**is** $\neg \text{decidable } ?K$)

proof

assume *decidable* *?K*

then obtain *f* **where** *recfn 1 f* **and** *f*: $\forall x. \text{eval } f [x] \downarrow = (\text{if } x \in ?K \text{ then } 1 \text{ else } 0)$

using *decidable-def* **by** *auto*

define *g* **where** *g* $\equiv \text{Cn } 1 \text{ r-ifeq-else-diverg } [f, Z, Z]$

then have *recfn 1 g*

using $\langle \text{recfn } 1 f \rangle$ *r-ifeq-else-diverg-recfn* **by** *simp*

with *assms* **obtain** *i* **where** *i*: *code i = g* **by** *auto*

from *g-def* **have** *eval g [x] = (if x \notin ?K then Some 0 else None)* **for** *x*

using *r-ifeq-else-diverg-recfn* $\langle \text{recfn } 1 f \rangle$ *f* **by** *simp*

then have *eval g [i] $\downarrow \longleftrightarrow i \notin ?K$* **by** *simp*

also have $\dots \longleftrightarrow \text{eval } (\text{code } i) [i] \uparrow$ **by** *simp*

also have $\dots \longleftrightarrow \text{eval } g [i] \uparrow$

using *i* **by** *simp*

finally have *eval g [i] $\downarrow \longleftrightarrow \text{eval } g [i] \uparrow$* .

then show *False* **by** *auto*

qed

1.4 Encoding tuples and lists

This section is based on the Cantor encoding for pairs. Tuples are encoded by repeated application of the pairing function, lists by pairing their length with the code for a tuple. Thus tuples have a fixed length that must be known when decoding, whereas lists are dynamically sized and know their current length.

1.4.1 Pairs and tuples

The Cantor pairing function

definition $r\text{-triangle} \equiv r\text{-shrink } (Pr\ 1\ Z\ (r\text{-dummy } 1\ (Cn\ 2\ S\ [r\text{-add}])))$

lemma $r\text{-triangle-prim}$: $prim\text{-recfn } 1\ r\text{-triangle}$
unfolding $r\text{-triangle-def}$ **by** $simp$

lemma $r\text{-triangle}$: $eval\ r\text{-triangle } [n] \downarrow = Sum\ \{0..n\}$

proof –

let $?r = r\text{-dummy } 1\ (Cn\ 2\ S\ [r\text{-add}])$
have $eval\ ?r\ [x, y, z] \downarrow = Suc\ (x + y)$ **for** $x\ y\ z$
using $r\text{-dummy-append}$ [of $Cn\ 2\ S\ [r\text{-add}]$ $[x, y]$ $[z]$ 1] **by** $simp$
then have $eval\ (Pr\ 1\ Z\ ?r)\ [x, y] \downarrow = Sum\ \{0..x\}$ **for** $x\ y$
by ($induction\ x$) $simp\text{-all}$
then show $?thesis$ **unfolding** $r\text{-triangle-def}$ **by** $simp$

qed

lemma $r\text{-triangle-eq-triangle}$ [$simp$]: $eval\ r\text{-triangle } [n] \downarrow = triangle\ n$
using $r\text{-triangle\ gauss-sum-nat\ triangle-def}$ **by** $simp$

definition $r\text{-prod-encode} \equiv Cn\ 2\ r\text{-add } [Cn\ 2\ r\text{-triangle } [r\text{-add}], Id\ 2\ 0]$

lemma $r\text{-prod-encode-prim}$ [$simp$]: $prim\text{-recfn } 2\ r\text{-prod-encode}$
unfolding $r\text{-prod-encode-def}$ **using** $r\text{-triangle-prim}$ **by** $simp$

lemma $r\text{-prod-encode}$ [$simp$]: $eval\ r\text{-prod-encode } [m, n] \downarrow = prod\text{-encode } (m, n)$
unfolding $r\text{-prod-encode-def}$ $prod\text{-encode-def}$ **using** $r\text{-triangle-prim}$ **by** $simp$

These abbreviations are just two more things borrowed from Xu et al. [18].

abbreviation $pdec1\ z \equiv fst\ (prod\text{-decode } z)$

abbreviation $pdec2\ z \equiv snd\ (prod\text{-decode } z)$

lemma $pdec1\text{-le}$: $pdec1\ i \leq i$
by ($metis\ le\text{-prod-encode-1}\ prod.\text{collapse}\ prod\text{-decode-inverse}$)

lemma $pdec2\text{-le}$: $pdec2\ i \leq i$
by ($metis\ le\text{-prod-encode-2}\ prod.\text{collapse}\ prod\text{-decode-inverse}$)

lemma $pdec\text{-less}$: $pdec2\ i < Suc\ i$
using $pdec2\text{-le}$ **by** ($simp\ add:\ le\text{-imp-less-Suc}$)

lemma $pdec1\text{-zero}$: $pdec1\ 0 = 0$
using $pdec1\text{-le}$ **by** $auto$

definition $r\text{-maxletr} \equiv$
 $Pr\ 1\ Z\ (Cn\ 3\ r\text{-ifte } [r\text{-dummy } 2\ (Cn\ 1\ r\text{-triangle } [S]), Id\ 3\ 2, Cn\ 3\ S\ [Id\ 3\ 0], Id\ 3\ 1])$

lemma $r\text{-maxletr-prim}$: $prim\text{-recfn } 2\ r\text{-maxletr}$
unfolding $r\text{-maxletr-def}$ **using** $r\text{-triangle-prim}$ **by** $simp$

lemma $not\text{-Suc-Greatest-not-Suc}$:
assumes $\neg P\ (Suc\ x)$ **and** $\exists x. P\ x$
shows $(GREATEST\ y. y \leq x \wedge P\ y) = (GREATEST\ y. y \leq Suc\ x \wedge P\ y)$
using $assms$ **by** ($metis\ le\text{-SucI}\ le\text{-Suc-eq}$)

lemma *r-maxletr*: $\text{eval } r\text{-maxletr } [x_0, x_1] \Downarrow = (\text{GREATEST } y. y \leq x_0 \wedge \text{triangle } y \leq x_1)$
proof –
let $?g = \text{Cn } 3 \text{ r-ifle } [r\text{-dummy } 2 (\text{Cn } 1 \text{ r-triangle } [S]), \text{Id } 3 \ 2, \text{Cn } 3 \ S \ [\text{Id } 3 \ 0], \text{Id } 3 \ 1]$
have *greatest*:
(if triangle (Suc x₀) ≤ x₁ then Suc x₀ else (GREATEST y. y ≤ x₀ ∧ triangle y ≤ x₁)) = (GREATEST y. y ≤ Suc x₀ ∧ triangle y ≤ x₁)
for $x_0 \ x_1$
proof *(cases triangle (Suc x₀) ≤ x₁)*
case *True*
then show *?thesis*
using *Greatest-equality*[of $\lambda y. y \leq \text{Suc } x_0 \wedge \text{triangle } y \leq x_1$] **by** *fastforce*
next
case *False*
then show *?thesis*
using *not-Suc-Greatest-not-Suc*[of $\lambda y. \text{triangle } y \leq x_1 \ x_0$] **by** *fastforce*
qed
show *?thesis*
unfolding *r-maxletr-def* **using** *r-triangle-prim*
proof *(induction x₀)*
case *0*
then show *?case*
using *Greatest-equality*[of $\lambda y. y \leq 0 \wedge \text{triangle } y \leq x_1 \ 0$] **by** *simp*
next
case *(Suc x₀)*
then show *?case* **using** *greatest* **by** *simp*
qed
qed

definition *r-maxlt* $\equiv r\text{-shrink } r\text{-maxletr}$

lemma *r-maxlt-prim*: *prim-recfn 1 r-maxlt*
unfolding *r-maxlt-def* **using** *r-maxletr-prim* **by** *simp*

lemma *r-maxlt*: $\text{eval } r\text{-maxlt } [e] \Downarrow = (\text{GREATEST } y. \text{triangle } y \leq e)$
proof –
have $y \leq \text{triangle } y$ **for** y
by *(induction y) auto*
then have $\text{triangle } y \leq e \implies y \leq e$ **for** $y \ e$
using *order-trans* **by** *blast*
then have $(\text{GREATEST } y. y \leq e \wedge \text{triangle } y \leq e) = (\text{GREATEST } y. \text{triangle } y \leq e)$
by *metis*
moreover have $\text{eval } r\text{-maxlt } [e] \Downarrow = (\text{GREATEST } y. y \leq e \wedge \text{triangle } y \leq e)$
using *r-maxletr r-shrink r-maxlt-def r-maxletr-prim* **by** *fastforce*
ultimately show *?thesis* **by** *simp*
qed

definition *pdec1'* $e \equiv e - \text{triangle } (\text{GREATEST } y. \text{triangle } y \leq e)$

definition *pdec2'* $e \equiv (\text{GREATEST } y. \text{triangle } y \leq e) - pdec1' \ e$

lemma *max-triangle-bound*: $\text{triangle } z \leq e \implies z \leq e$
by *(metis Suc-pred add-leD2 less-Suc-eq triangle-Suc zero-le zero-less-Suc)*

lemma *triangle-greatest-le*: $\text{triangle } (\text{GREATEST } y. \text{triangle } y \leq e) \leq e$
using *max-triangle-bound GreatestI-nat*[of $\lambda y. \text{triangle } y \leq e \ 0 \ e$] **by** *simp*

lemma *prod-encode-pdec'*: $\text{prod-encode } (pdec1' e, pdec2' e) = e$
proof –
 let $?P = \lambda y. \text{triangle } y \leq e$
 let $?y = \text{GREATEST } y. ?P y$
 have $pdec1' e \leq ?y$
proof (*rule ccontr*)
 assume $\neg pdec1' e \leq ?y$
 then have $e - \text{triangle } ?y > ?y$
 using *pdec1'-def* **by** *simp*
 then have $?P (\text{Suc } ?y)$ **by** *simp*
 moreover have $\forall z. ?P z \longrightarrow z \leq e$
 using *max-triangle-bound* **by** *simp*
 ultimately have $\text{Suc } ?y \leq ?y$
 using *Greatest-le-nat*[of $?P \text{Suc } ?y e$] **by** *blast*
 then show *False* **by** *simp*
 qed
 then have $pdec1' e + pdec2' e = ?y$
 using *pdec1'-def pdec2'-def* **by** *simp*
 then have $\text{prod-encode } (pdec1' e, pdec2' e) = \text{triangle } ?y + pdec1' e$
by (*simp add: prod-encode-def*)
 then show *?thesis* **using** *pdec1'-def triangle-greatest-le* **by** *simp*
 qed

lemma *pdec'*:
 $pdec1' e = pdec1 e$
 $pdec2' e = pdec2 e$
using *prod-encode-pdec' prod-encode-inverse* **by** (*metis fst-conv, metis snd-conv*)

definition *r-pdec1* $\equiv Cn\ 1\ r\text{-sub } [Id\ 1\ 0, Cn\ 1\ r\text{-triangle } [r\text{-maxlt}]]$

lemma *r-pdec1-prim* [*simp*]: *prim-recfn* 1 *r-pdec1*
unfolding *r-pdec1-def* **using** *r-triangle-prim r-maxlt-prim* **by** *simp*

lemma *r-pdec1* [*simp*]: *eval* *r-pdec1* [e] $\downarrow = pdec1 e$
unfolding *r-pdec1-def* **using** *r-triangle-prim r-maxlt-prim pdec' pdec1'-def*
by (*simp add: r-maxlt*)

definition *r-pdec2* $\equiv Cn\ 1\ r\text{-sub } [r\text{-maxlt}, r\text{-pdec1}]$

lemma *r-pdec2-prim* [*simp*]: *prim-recfn* 1 *r-pdec2*
unfolding *r-pdec2-def* **using** *r-maxlt-prim* **by** *simp*

lemma *r-pdec2* [*simp*]: *eval* *r-pdec2* [e] $\downarrow = pdec2 e$
unfolding *r-pdec2-def* **using** *r-maxlt-prim r-maxlt pdec' pdec2'-def* **by** *simp*

abbreviation *pdec12* $i \equiv pdec1 (pdec2 i)$
abbreviation *pdec22* $i \equiv pdec2 (pdec2 i)$
abbreviation *pdec122* $i \equiv pdec1 (pdec22 i)$
abbreviation *pdec222* $i \equiv pdec2 (pdec22 i)$

definition *r-pdec12* $\equiv Cn\ 1\ r\text{-pdec1 } [r\text{-pdec2}]$

lemma *r-pdec12-prim* [*simp*]: *prim-recfn* 1 *r-pdec12*
unfolding *r-pdec12-def* **by** *simp*

lemma *r-pdec12* [simp]: *eval r-pdec12 [e] ↓ = pdec12 e*
unfolding *r-pdec12-def* **by** *simp*

definition *r-pdec22* \equiv *Cn 1 r-pdec2 [r-pdec2]*

lemma *r-pdec22-prim* [simp]: *prim-recfn 1 r-pdec22*
unfolding *r-pdec22-def* **by** *simp*

lemma *r-pdec22* [simp]: *eval r-pdec22 [e] ↓ = pdec22 e*
unfolding *r-pdec22-def* **by** *simp*

definition *r-pdec122* \equiv *Cn 1 r-pdec1 [r-pdec22]*

lemma *r-pdec122-prim* [simp]: *prim-recfn 1 r-pdec122*
unfolding *r-pdec122-def* **by** *simp*

lemma *r-pdec122* [simp]: *eval r-pdec122 [e] ↓ = pdec122 e*
unfolding *r-pdec122-def* **by** *simp*

definition *r-pdec222* \equiv *Cn 1 r-pdec2 [r-pdec22]*

lemma *r-pdec222-prim* [simp]: *prim-recfn 1 r-pdec222*
unfolding *r-pdec222-def* **by** *simp*

lemma *r-pdec222* [simp]: *eval r-pdec222 [e] ↓ = pdec222 e*
unfolding *r-pdec222-def* **by** *simp*

The Cantor tuple function

The empty tuple gets no code, whereas singletons are encoded by their only element and other tuples by recursively applying the pairing function. This yields, for every n , the function *tuple-encode n*, which is a bijection between the natural numbers and the lists of length $(n + 1)$.

fun *tuple-encode* :: *nat* \Rightarrow *nat list* \Rightarrow *nat* **where**
tuple-encode n [] = undefined
| *tuple-encode 0 (x # xs) = x*
| *tuple-encode (Suc n) (x # xs) = prod-encode (x, tuple-encode n xs)*

lemma *tuple-encode-prod-encode*: *tuple-encode 1 [x, y] = prod-encode (x, y)*
by *simp*

fun *tuple-decode* **where**
tuple-decode 0 i = [i]
| *tuple-decode (Suc n) i = pdec1 i # tuple-decode n (pdec2 i)*

lemma *tuple-encode-decode* [simp]:
tuple-encode (length xs - 1) (tuple-decode (length xs - 1) i) = i

proof (*induction length xs - 1 arbitrary: xs i*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have *length xs - 1 > 0* **by** *simp*
with *Suc* **have** *: *tuple-encode n (tuple-decode n j) = j* **for** *j*
by (*metis diff-Suc-1 length-tl*)

from *Suc* **have** *tuple-decode* (*Suc n*) *i* = *pdec1 i # tuple-decode n (pdec2 i)*
using *tuple-decode.simps(2)* **by** *blast*
then have *tuple-encode* (*Suc n*) (*tuple-decode* (*Suc n*) *i*) =
tuple-encode (*Suc n*) (*pdec1 i # tuple-decode n (pdec2 i)*)
using *Suc* **by** *simp*
also have ... = *prod-encode* (*pdec1 i, tuple-encode n (tuple-decode n (pdec2 i))*)
by *simp*
also have ... = *prod-encode* (*pdec1 i, pdec2 i*)
using *Suc ** **by** *simp*
also have ... = *i* **by** *simp*
finally have *tuple-encode* (*Suc n*) (*tuple-decode* (*Suc n*) *i*) = *i* .
then show ?*case* **by** (*simp add: Suc.hyps(2)*)
qed

lemma *tuple-encode-decode'* [*simp*]: *tuple-encode n (tuple-decode n i) = i*
using *tuple-encode-decode* **by** (*metis Ex-list-of-length diff-Suc-1 length-Cons*)

lemma *tuple-decode-encode*:
assumes *length xs > 0*
shows *tuple-decode* (*length xs - 1*) (*tuple-encode* (*length xs - 1*) *xs*) = *xs*
using *assms*

proof (*induction length xs - 1 arbitrary: xs*)
case 0
moreover from this have *length xs = 1* **by** *linarith*
ultimately show ?*case*
by (*metis One-nat-def length-0-conv length-Suc-conv tuple-decode.simps(1)*
tuple-encode.simps(2))

next
case (*Suc n*)
let ?*t* = *tl xs*
let ?*i* = *tuple-encode* (*Suc n*) *xs*
have *length ?t > 0* **and** *length ?t - 1 = n*
using *Suc* **by** *simp-all*
then have *tuple-decode n (tuple-encode n ?t) = ?t*
using *Suc* **by** *blast*
moreover have ?*i* = *prod-encode* (*hd xs, tuple-encode n ?t*)
using *Suc* **by** (*metis hd-Cons-tl length-greater-0-conv tuple-encode.simps(3)*)
moreover have *tuple-decode* (*Suc n*) ?*i* = *pdec1 ?i # tuple-decode n (pdec2 ?i)*
using *tuple-decode.simps(2)* **by** *blast*
ultimately have *tuple-decode* (*Suc n*) ?*i* = *xs*
using *Suc.prem*s **by** *simp*
then show ?*case* **by** (*simp add: Suc.hyps(2)*)
qed

lemma *tuple-decode-encode'* [*simp*]:
assumes *length xs = Suc n*
shows *tuple-decode n (tuple-encode n xs) = xs*
using *assms tuple-decode-encode* **by** (*metis diff-Suc-1 zero-less-Suc*)

lemma *tuple-decode-length* [*simp*]: *length (tuple-decode n i) = Suc n*
by (*induction n arbitrary: i*) *simp-all*

lemma *tuple-decode-nonzero*:
assumes *n > 0*
shows *tuple-decode n i = pdec1 i # tuple-decode (n - 1) (pdec2 i)*
using *assms* **by** (*metis One-nat-def Suc-pred tuple-decode.simps(2)*)

The tuple encoding functions are primitive recursive.

```
fun r-tuple-encode :: nat ⇒ recf where
  r-tuple-encode 0 = Id 1 0
| r-tuple-encode (Suc n) =
  Cn (Suc (Suc n)) r-prod-encode [Id (Suc (Suc n)) 0, r-shift (r-tuple-encode n)]
```

```
lemma r-tuple-encode-prim [simp]: prim-recfn (Suc n) (r-tuple-encode n)
by (induction n) simp-all
```

```
lemma r-tuple-encode:
assumes length xs = Suc n
shows eval (r-tuple-encode n) xs ↓= tuple-encode n xs
using assms
proof (induction n arbitrary: xs)
case 0
then show ?case
by (metis One-nat-def eval-Id length-Suc-conv nth-Cons-0
  r-tuple-encode.simps(1) tuple-encode.simps(2) zero-less-one)
next
case (Suc n)
then obtain y ys where y-ys: y # ys = xs
by (metis length-Suc-conv)
with Suc have eval (r-tuple-encode n) ys ↓= tuple-encode n ys
by auto
with y-ys have eval (r-shift (r-tuple-encode n)) xs ↓= tuple-encode n ys
using Suc.prems r-shift-prim r-tuple-encode-prim by auto
moreover have eval (Id (Suc (Suc n)) 0) xs ↓= y
using y-ys Suc.prems by auto
ultimately have eval (r-tuple-encode (Suc n)) xs ↓= prod-encode (y, tuple-encode n ys)
using Suc.prems by simp
then show ?case using y-ys by auto
qed
```

Functions on encoded tuples

The function for accessing the n -th element of a tuple returns 0 for out-of-bounds access.

```
definition e-tuple-nth :: nat ⇒ nat ⇒ nat ⇒ nat where
  e-tuple-nth a i n ≡ if  $n \leq a$  then (tuple-decode a i) ! n else 0
```

```
lemma e-tuple-nth-le [simp]:  $n \leq a \implies e\text{-tuple-nth } a \ i \ n = (\text{tuple-decode } a \ i) \ ! \ n$ 
using e-tuple-nth-def by simp
```

```
lemma e-tuple-nth-gr [simp]:  $n > a \implies e\text{-tuple-nth } a \ i \ n = 0$ 
using e-tuple-nth-def by simp
```

```
lemma tuple-decode-pdec2: tuple-decode a (pdec2 es) = tl (tuple-decode (Suc a) es)
by simp
```

```
fun iterate :: nat ⇒ ('a ⇒ 'a) ⇒ ('a ⇒ 'a) where
  iterate 0 f = id
| iterate (Suc n) f = f ∘ (iterate n f)
```

```
lemma iterate-additive:
assumes iterate t1 f x = y and iterate t2 f y = z
shows iterate (t1 + t2) f x = z
```

using *assms* **by** (*induction t₂ arbitrary: z*) *auto*

lemma *iterate-additive'*: $\text{iterate } (t_1 + t_2) f x = \text{iterate } t_2 f (\text{iterate } t_1 f x)$
using *iterate-additive* **by** *metis*

lemma *e-tuple-nth-elementary*:

assumes $k \leq a$

shows $e\text{-tuple-nth } a \ i \ k = (\text{if } a = k \text{ then } (\text{iterate } k \text{ pdec2 } i) \text{ else } (\text{pdec1 } (\text{iterate } k \text{ pdec2 } i)))$

proof –

have *: $\text{tuple-decode } (a - k) (\text{iterate } k \text{ pdec2 } i) = \text{drop } k (\text{tuple-decode } a \ i)$

using *assms*

by (*induction k*) (*simp, simp add: Suc-diff-Suc tuple-decode-pdec2 drop-Suc tl-drop*)

show *?thesis*

proof (*cases a = k*)

case *True*

then have $\text{tuple-decode } 0 (\text{iterate } k \text{ pdec2 } i) = \text{drop } k (\text{tuple-decode } a \ i)$

using *assms* * **by** *simp*

moreover from *this* **have** $\text{drop } k (\text{tuple-decode } a \ i) = [\text{tuple-decode } a \ i \ ! \ k]$

using *assms True* **by** (*metis nth-via-drop tuple-decode.simps(1)*)

ultimately show *?thesis* **using** *True* **by** *simp*

next

case *False*

with *assms* **have** $a - k > 0$ **by** *simp*

with * **have** $\text{tuple-decode } (a - k) (\text{iterate } k \text{ pdec2 } i) = \text{drop } k (\text{tuple-decode } a \ i)$

by *simp*

then have $\text{pdec1 } (\text{iterate } k \text{ pdec2 } i) = \text{hd } (\text{drop } k (\text{tuple-decode } a \ i))$

using *tuple-decode-nonzero* $\langle a - k > 0 \rangle$ **by** (*metis list.sel(1)*)

with $\langle a - k > 0 \rangle$ **have** $\text{pdec1 } (\text{iterate } k \text{ pdec2 } i) = (\text{tuple-decode } a \ i) \ ! \ k$

by (*simp add: hd-drop-conv-nth*)

with *False assms* **show** *?thesis* **by** *simp*

qed

qed

definition *r-nth-inbounds* \equiv

let $r = \text{Pr } 1 \ (\text{Id } 1 \ 0) \ (\text{Cn } 3 \ r\text{-pdec2 } [\text{Id } 3 \ 1])$

in $\text{Cn } 3 \ r\text{-ifeq}$

$[\text{Id } 3 \ 0,$

$\text{Id } 3 \ 2,$

$\text{Cn } 3 \ r \ [\text{Id } 3 \ 2, \ \text{Id } 3 \ 1],$

$\text{Cn } 3 \ r\text{-pdec1 } [\text{Cn } 3 \ r \ [\text{Id } 3 \ 2, \ \text{Id } 3 \ 1]]]$

lemma *r-nth-inbounds-prim*: *prim-recfn 3 r-nth-inbounds*

unfolding *r-nth-inbounds-def* **by** (*simp add: Let-def*)

lemma *r-nth-inbounds*:

$k \leq a \implies \text{eval } r\text{-nth-inbounds } [a, i, k] \downarrow = e\text{-tuple-nth } a \ i \ k$

$\text{eval } r\text{-nth-inbounds } [a, i, k] \downarrow$

proof –

let $?r = \text{Pr } 1 \ (\text{Id } 1 \ 0) \ (\text{Cn } 3 \ r\text{-pdec2 } [\text{Id } 3 \ 1])$

let $?h = \text{Cn } 3 \ ?r \ [\text{Id } 3 \ 2, \ \text{Id } 3 \ 1]$

have $\text{eval } ?r \ [k, i] \downarrow = \text{iterate } k \text{ pdec2 } i$ **for** $k \ i$

using *r-pdec2-prim* **by** (*induction k*) (*simp-all*)

then have $\text{eval } ?h \ [a, i, k] \downarrow = \text{iterate } k \text{ pdec2 } i$

using *r-pdec2-prim* **by** *simp*

then have $\text{eval } r\text{-nth-inbounds } [a, i, k] \downarrow =$

$(\text{if } a = k \text{ then } \text{iterate } k \text{ pdec2 } i \text{ else } \text{pdec1 } (\text{iterate } k \text{ pdec2 } i))$

```

  unfolding r-nth-inbounds-def by (simp add: Let-def)
then show  $k \leq a \implies \text{eval } r\text{-nth-inbounds } [a, i, k] \Downarrow = e\text{-tuple-nth } a \ i \ k$ 
  and  $\text{eval } r\text{-nth-inbounds } [a, i, k] \Downarrow$ 
  using e-tuple-nth-elementary by simp-all
qed

```

```

definition r-tuple-nth  $\equiv$ 
  Cn 3 r-ifle [Id 3 2, Id 3 0, r-nth-inbounds, r-constn 2 0]

```

```

lemma r-tuple-nth-prim: prim-recfn 3 r-tuple-nth
  unfolding r-tuple-nth-def using r-nth-inbounds-prim by simp

```

```

lemma r-tuple-nth [simp]:  $\text{eval } r\text{-tuple-nth } [a, i, k] \Downarrow = e\text{-tuple-nth } a \ i \ k$ 
  unfolding r-tuple-nth-def using r-nth-inbounds-prim r-nth-inbounds by simp

```

1.4.2 Lists

Encoding and decoding

Lists are encoded by pairing the length of the list with the code for the tuple made up of the list's elements. Then all these codes are incremented in order to make room for the empty list (cf. Rogers [12, p. 71]).

```

fun list-encode :: nat list  $\Rightarrow$  nat where
  list-encode [] = 0
| list-encode (x # xs) = Suc (prod-encode (length xs, tuple-encode (length xs) (x # xs)))

```

```

lemma list-encode-0 [simp]:  $\text{list-encode } xs = 0 \iff xs = []$ 
  using list-encode.elims Partial-Recursive.list-encode.simps(1) by blast

```

```

lemma list-encode-1:  $\text{list-encode } [0] = 1$ 
  by (simp add: prod-encode-def)

```

```

fun list-decode :: nat  $\Rightarrow$  nat list where
  list-decode 0 = []
| list-decode (Suc n) = tuple-decode (pdec1 n) (pdec2 n)

```

```

lemma list-encode-decode [simp]:  $\text{list-encode } (\text{list-decode } n) = n$ 

```

```

proof (cases n)

```

```

  case 0

```

```

  then show ?thesis by simp

```

```

next

```

```

  case (Suc k)

```

```

  then have *:  $\text{list-decode } n = \text{tuple-decode } (\text{pdec1 } k) (\text{pdec2 } k)$  (is - = ?t)

```

```

  by simp

```

```

  then obtain x xs where  $x \# xs = ?t$ 

```

```

  by (metis tuple-decode.elims)

```

```

  then have list-encode ?t =  $\text{list-encode } (x \# xs)$  by simp

```

```

  then have 1:  $\text{list-encode } ?t = \text{Suc } (\text{prod-encode } (\text{length } xs, \text{tuple-encode } (\text{length } xs) (x \# xs)))$ 

```

```

  by simp

```

```

  have 2:  $\text{length } xs = \text{length } ?t - 1$ 

```

```

  using xs by (metis length-tl list.sel(3))

```

```

  then have 3:  $\text{length } xs = \text{pdec1 } k$ 

```

```

  using * by simp

```

```

  then have  $\text{tuple-encode } (\text{length } ?t - 1) ?t = \text{pdec2 } k$ 

```

```

  using 2 tuple-encode-decode by metis

```

then have $list\text{-}encode\ ?t = Suc\ (prod\text{-}encode\ (pdec1\ k,\ pdec2\ k))$
using $1\ 2\ 3\ xxs$ **by** $simp$
with $*\ Suc$ **show** $?thesis$ **by** $simp$
qed

lemma $list\text{-}decode\text{-}encode$ [$simp$]: $list\text{-}decode\ (list\text{-}encode\ xs) = xs$

proof ($cases\ xs$)

case Nil

then show $?thesis$ **by** $simp$

next

case ($Cons\ y\ ys$)

then have $list\text{-}encode\ xs =$

$Suc\ (prod\text{-}encode\ (length\ ys,\ tuple\text{-}encode\ (length\ ys)\ xs))$

($is\ - = Suc\ ?i$)

by $simp$

then have $list\text{-}decode\ (Suc\ ?i) = tuple\text{-}decode\ (pdec1\ ?i)\ (pdec2\ ?i)$ **by** $simp$

moreover have $pdec1\ ?i = length\ ys$ **by** $simp$

moreover have $pdec2\ ?i = tuple\text{-}encode\ (length\ ys)\ xs$ **by** $simp$

ultimately have $list\text{-}decode\ (Suc\ ?i) =$

$tuple\text{-}decode\ (length\ ys)\ (tuple\text{-}encode\ (length\ ys)\ xs)$

by $simp$

moreover have $length\ ys = length\ xs - 1$

using $Cons$ **by** $simp$

ultimately have $list\text{-}decode\ (Suc\ ?i) =$

$tuple\text{-}decode\ (length\ xs - 1)\ (tuple\text{-}encode\ (length\ xs - 1)\ xs)$

by $simp$

then show $?thesis$ **using** $Cons$ **by** $simp$

qed

abbreviation $singleton\text{-}encode :: nat \Rightarrow nat$ **where**

$singleton\text{-}encode\ x \equiv list\text{-}encode\ [x]$

lemma $list\text{-}decode\text{-}singleton$: $list\text{-}decode\ (singleton\text{-}encode\ x) = [x]$

by $simp$

definition $r\text{-}singleton\text{-}encode \equiv Cn\ 1\ S\ [Cn\ 1\ r\text{-}prod\text{-}encode\ [Z,\ Id\ 1\ 0]]$

lemma $r\text{-}singleton\text{-}encode\text{-}prim$ [$simp$]: $prim\text{-}recfn\ 1\ r\text{-}singleton\text{-}encode$

unfolding $r\text{-}singleton\text{-}encode\text{-}def$ **by** $simp$

lemma $r\text{-}singleton\text{-}encode$ [$simp$]: $eval\ r\text{-}singleton\text{-}encode\ [x] \Downarrow = singleton\text{-}encode\ x$

unfolding $r\text{-}singleton\text{-}encode\text{-}def$ **by** $simp$

definition $r\text{-}list\text{-}encode :: nat \Rightarrow recf$ **where**

$r\text{-}list\text{-}encode\ n \equiv Cn\ (Suc\ n)\ S\ [Cn\ (Suc\ n)\ r\text{-}prod\text{-}encode\ [r\text{-}constn\ n\ n,\ r\text{-}tuple\text{-}encode\ n]]$

lemma $r\text{-}list\text{-}encode\text{-}prim$ [$simp$]: $prim\text{-}recfn\ (Suc\ n)\ (r\text{-}list\text{-}encode\ n)$

unfolding $r\text{-}list\text{-}encode\text{-}def$ **by** $simp$

lemma $r\text{-}list\text{-}encode$:

assumes $length\ xs = Suc\ n$

shows $eval\ (r\text{-}list\text{-}encode\ n)\ xs \Downarrow = list\text{-}encode\ xs$

proof -

have $eval\ (r\text{-}tuple\text{-}encode\ n)\ xs \Downarrow$

by ($simp\ add: assms\ r\text{-}tuple\text{-}encode$)

then have $eval\ (Cn\ (Suc\ n)\ r\text{-}prod\text{-}encode\ [r\text{-}constn\ n\ n,\ r\text{-}tuple\text{-}encode\ n])\ xs \Downarrow$

```

    using assms by simp
  then have eval (r-list-encode n) xs =
    eval S [the (eval (Cn (Suc n) r-prod-encode [r-constn n n, r-tuple-encode n]) xs)]
    unfolding r-list-encode-def using assms r-tuple-encode by simp
  moreover from assms obtain y ys where xs = y # ys
    by (meson length-Suc-conv)
  ultimately show ?thesis
    unfolding r-list-encode-def using assms r-tuple-encode by simp
qed

```

Functions on encoded lists

The functions in this section mimic those on type *nat list*. Their names are prefixed by *e-* and the names of the corresponding *recfs* by *r-*.

abbreviation *e-tl* :: *nat* ⇒ *nat* **where**
e-tl *e* ≡ *list-encode* (*tl* (*list-decode* *e*))

In order to turn *e-tl* into a partial recursive function we first represent it in a more elementary way.

lemma *e-tl-elementary*:

```

e-tl e =
  (if e = 0 then 0
   else if pdec1 (e - 1) = 0 then 0
   else Suc (prod-encode (pdec1 (e - 1) - 1, pdec22 (e - 1))))

```

proof (*cases* *e*)

```

  case 0
  then show ?thesis by simp
next
  case Suc-d: (Suc d)
  then show ?thesis
  proof (cases pdec1 d)
    case 0
    then show ?thesis using Suc-d by simp

```

```

next
  case (Suc a)
  have *: list-decode e = tuple-decode (pdec1 d) (pdec2 d)
    using Suc-d by simp
  with Suc obtain x xs where xs: list-decode e = x # xs by simp
  then have **: e-tl e = list-encode xs by simp
  have list-decode (Suc (prod-encode (pdec1 (e - 1) - 1, pdec22 (e - 1)))) =
    tuple-decode (pdec1 (e - 1) - 1) (pdec22 (e - 1))
    (is ?lhs = -)
    by simp
  also have ... = tuple-decode a (pdec22 (e - 1))
    using Suc Suc-d by simp
  also have ... = tl (tuple-decode (Suc a) (pdec2 (e - 1)))
    using tuple-decode-pdec2 Suc by presburger
  also have ... = tl (tuple-decode (pdec1 (e - 1)) (pdec2 (e - 1)))
    using Suc Suc-d by auto
  also have ... = tl (list-decode e)
    using * Suc-d by simp
  also have ... = xs
    using xs by simp
  finally have ?lhs = xs .
  then have list-encode ?lhs = list-encode xs by simp

```

then have Suc ($prod\text{-}encode$ ($pdec1$ ($e - 1$) - 1, $pdec22$ ($e - 1$))) = $list\text{-}encode$ xs
using $list\text{-}encode\text{-}decode$ **by** $metis$
then show $?thesis$ **using** $**$ $Suc\text{-}d$ Suc **by** $simp$
qed
qed

definition $r\text{-}tl$ \equiv
 let $r = Cn$ 1 $r\text{-}pdec1$ [$r\text{-}dec$]
 in Cn 1 $r\text{-}ifz$
 $[Id$ 1 0,
 $Z,$
 Cn 1 $r\text{-}ifz$
 $[r, Z, Cn$ 1 S [Cn 1 $r\text{-}prod\text{-}encode$ [Cn 1 $r\text{-}dec$ [r], Cn 1 $r\text{-}pdec22$ [$r\text{-}dec$]]]]]

lemma $r\text{-}tl\text{-}prim$ [$simp$]: $prim\text{-}recfn$ 1 $r\text{-}tl$
unfolding $r\text{-}tl\text{-}def$ **by** ($simp$ add : $Let\text{-}def$)

lemma $r\text{-}tl$ [$simp$]: $eval$ $r\text{-}tl$ [e] $\Downarrow = e\text{-}tl$ e
unfolding $r\text{-}tl\text{-}def$ **using** $e\text{-}tl\text{-}elementary$ **by** ($simp$ add : $Let\text{-}def$)

We define the head of the empty encoded list to be zero.

definition $e\text{-}hd$ $:: nat \Rightarrow nat$ **where**
 $e\text{-}hd$ $e \equiv if$ $e = 0$ $then$ 0 $else$ hd ($list\text{-}decode$ e)

lemma $e\text{-}hd$ [$simp$]:
assumes $list\text{-}decode$ $e = x \# xs$
shows $e\text{-}hd$ $e = x$
using $e\text{-}hd\text{-}def$ $assms$ **by** $auto$

lemma $e\text{-}hd\text{-}0$ [$simp$]: $e\text{-}hd$ 0 = 0
using $e\text{-}hd\text{-}def$ **by** $simp$

lemma $e\text{-}hd\text{-}neq\text{-}0$ [$simp$]:
assumes $e \neq 0$
shows $e\text{-}hd$ $e = hd$ ($list\text{-}decode$ e)
using $e\text{-}hd\text{-}def$ $assms$ **by** $simp$

definition $r\text{-}hd$ \equiv
 Cn 1 $r\text{-}ifz$ [Cn 1 $r\text{-}pdec1$ [$r\text{-}dec$], Cn 1 $r\text{-}pdec2$ [$r\text{-}dec$], Cn 1 $r\text{-}pdec12$ [$r\text{-}dec$]]

lemma $r\text{-}hd\text{-}prim$ [$simp$]: $prim\text{-}recfn$ 1 $r\text{-}hd$
unfolding $r\text{-}hd\text{-}def$ **by** $simp$

lemma $r\text{-}hd$ [$simp$]: $eval$ $r\text{-}hd$ [e] $\Downarrow = e\text{-}hd$ e

proof –

have $e\text{-}hd$ $e = (if$ $pdec1$ ($e - 1$) = 0 $then$ $pdec2$ ($e - 1$) $else$ $pdec12$ ($e - 1$))

proof ($cases$ e)

case 0

then show $?thesis$ **using** $pdec1\text{-}zero$ $pdec2\text{-}le$ **by** $auto$

next

case (Suc d)

then show $?thesis$ **by** ($cases$ $pdec1$ d) ($simp\text{-}all$ add : $pdec1\text{-}zero$)

qed

then show $?thesis$ **unfolding** $r\text{-}hd\text{-}def$ **by** $simp$

qed

abbreviation $e\text{-length} :: \text{nat} \Rightarrow \text{nat}$ **where**
 $e\text{-length } e \equiv \text{length } (\text{list-decode } e)$

lemma $e\text{-length-0}$: $e\text{-length } e = 0 \implies e = 0$
by (*metis list-encode.simps(1) length-0-conv list-encode-decode*)

definition $r\text{-length} \equiv \text{Cn } 1 \text{ r-ifz } [\text{Id } 1 \ 0, Z, \text{Cn } 1 \ S \ [\text{Cn } 1 \ r\text{-pdec1 } [r\text{-dec}]]]$

lemma $r\text{-length-prim}$ [*simp*]: $\text{prim-recfn } 1 \ r\text{-length}$
unfolding $r\text{-length-def}$ **by** *simp*

lemma $r\text{-length}$ [*simp*]: $\text{eval } r\text{-length } [e] \Downarrow = e\text{-length } e$
unfolding $r\text{-length-def}$ **by** (*cases e*) *simp-all*

Accessing an encoded list out of bounds yields zero.

definition $e\text{-nth} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $e\text{-nth } e \ n \equiv \text{if } e = 0 \text{ then } 0 \text{ else } e\text{-tuple-nth } (pdec1 \ (e - 1)) \ (pdec2 \ (e - 1)) \ n$

lemma $e\text{-nth}$ [*simp*]:
 $e\text{-nth } e \ n = (\text{if } n < e\text{-length } e \text{ then } (\text{list-decode } e) ! \ n \text{ else } 0)$
by (*cases e*) (*simp-all add: e-nth-def e-tuple-nth-def*)

lemma $e\text{-hd-nth0}$: $e\text{-hd } e = e\text{-nth } e \ 0$
by (*simp add: e-hd-def e-length-0 hd-conv-nth*)

definition $r\text{-nth} \equiv$
 $\text{Cn } 2 \ r\text{-ifz}$
 $[\text{Id } 2 \ 0,$
 $r\text{-constn } 1 \ 0,$
 $\text{Cn } 2 \ r\text{-tuple-nth}$
 $[\text{Cn } 2 \ r\text{-pdec1 } [r\text{-dummy } 1 \ r\text{-dec}], \text{Cn } 2 \ r\text{-pdec2 } [r\text{-dummy } 1 \ r\text{-dec}], \text{Id } 2 \ 1]$

lemma $r\text{-nth-prim}$ [*simp*]: $\text{prim-recfn } 2 \ r\text{-nth}$
unfolding $r\text{-nth-def}$ **using** $r\text{-tuple-nth-prim}$ **by** *simp*

lemma $r\text{-nth}$ [*simp*]: $\text{eval } r\text{-nth } [e, n] \Downarrow = e\text{-nth } e \ n$
unfolding $r\text{-nth-def}$ $e\text{-nth-def}$ **using** $r\text{-tuple-nth-prim}$ **by** *simp*

definition $r\text{-rev-aux} \equiv$
 $\text{Pr } 1 \ r\text{-hd } (\text{Cn } 3 \ r\text{-prod-encode } [\text{Cn } 3 \ r\text{-nth } [\text{Id } 3 \ 2, \text{Cn } 3 \ S \ [\text{Id } 3 \ 0]], \text{Id } 3 \ 1])$

lemma $r\text{-rev-aux-prim}$: $\text{prim-recfn } 2 \ r\text{-rev-aux}$
unfolding $r\text{-rev-aux-def}$ **by** *simp*

lemma $r\text{-rev-aux}$:
assumes $\text{list-decode } e = xs$ **and** $\text{length } xs > 0$ **and** $i < \text{length } xs$
shows $\text{eval } r\text{-rev-aux } [i, e] \Downarrow = \text{tuple-encode } i \ (\text{rev } (\text{take } (\text{Suc } i) \ xs))$
using *assms(3)*
proof (*induction i*)
case 0
then show *?case*
unfolding $r\text{-rev-aux-def}$ **using** *assms e-hd-def r-hd* **by** (*auto simp add: take-Suc*)
next
case $(\text{Suc } i)$
let $?g = \text{Cn } 3 \ r\text{-prod-encode } [\text{Cn } 3 \ r\text{-nth } [\text{Id } 3 \ 2, \text{Cn } 3 \ S \ [\text{Id } 3 \ 0]], \text{Id } 3 \ 1]$
from Suc **have** $\text{eval } r\text{-rev-aux } [\text{Suc } i, e] = \text{eval } ?g \ [i, \text{the } (\text{eval } r\text{-rev-aux } [i, e]), e]$

unfolding *r-rev-aux-def* **by** *simp*
also have ... $\downarrow = \text{prod-encode } (xs \ ! \ (Suc \ i), \ \text{tuple-encode } \ i \ (\text{rev } (\text{take } \ (Suc \ i) \ xs)))$
using *Suc* **by** (*simp add: assms(1)*)
finally show *?case* **by** (*simp add: Suc.premis take-Suc-conv-app-nth*)
qed

corollary *r-rev-aux-full*:

assumes *list-decode e = xs* **and** *length xs > 0*
shows *eval r-rev-aux [length xs - 1, e]* $\downarrow = \text{tuple-encode } (\text{length } \ xs \ - \ 1) \ (\text{rev } \ xs)$
using *r-rev-aux assms* **by** *simp*

lemma *r-rev-aux-total*: *eval r-rev-aux [i, e]* \downarrow
using *r-rev-aux-prim totalE* **by** *fastforce*

definition *r-rev* \equiv

Cn 1 r-ifz
[Id 1 0,
Z,
Cn 1 S
[Cn 1 r-prod-encode
[Cn 1 r-dec [r-length], Cn 1 r-rev-aux [Cn 1 r-dec [r-length], Id 1 0]]]

lemma *r-rev-prim [simp]*: *prim-recfn 1 r-rev*
unfolding *r-rev-def* **using** *r-rev-aux-prim* **by** *simp*

lemma *r-rev [simp]*: *eval r-rev [e]* $\downarrow = \text{list-encode } (\text{rev } (\text{list-decode } \ e))$

proof –

let *?d = Cn 1 r-dec [r-length]*
let *?a = Cn 1 r-rev-aux [?d, Id 1 0]*
let *?p = Cn 1 r-prod-encode [?d, ?a]*
let *?s = Cn 1 S [?p]*
have *eval-a*: *eval ?a [e] = eval r-rev-aux [e-length e - 1, e]*
using *r-rev-aux-prim* **by** *simp*
then have *eval ?s [e]* \downarrow
using *r-rev-aux-prim* **by** (*simp add: r-rev-aux-total*)
then have *: *eval r-rev [e]* $\downarrow = (\text{if } \ e \ = \ 0 \ \text{then } \ 0 \ \text{else } \ \text{the } \ (\text{eval } \ ?s \ [e]))$
using *r-rev-aux-prim* **by** (*simp add: r-rev-def*)
show *?thesis*
proof (*cases e = 0*)
case *True*
then show *?thesis* **using** * **by** *simp*

next

case *False*
then obtain *xs* **where** *xs: xs = list-decode e length xs > 0*
using *e-length-0* **by** *auto*
then have *len*: *length xs = e-length e* **by** *simp*
with *eval-a* **have** *eval ?a [e] = eval r-rev-aux [length xs - 1, e]*
by *simp*
then have *eval ?a [e]* $\downarrow = \text{tuple-encode } (\text{length } \ xs \ - \ 1) \ (\text{rev } \ xs)$
using *xs r-rev-aux-full* **by** *simp*
then have *eval ?s [e]* $\downarrow =$
Suc (prod-encode (length xs - 1, tuple-encode (length xs - 1) (rev xs)))
using *len r-rev-aux-prim* **by** *simp*
then have *eval ?s [e]* $\downarrow =$
Suc (prod-encode
(length (rev xs) - 1, tuple-encode (length (rev xs) - 1) (rev xs)))

by *simp*
moreover have $\text{length } (\text{rev } xs) > 0$
using xs **by** *simp*
ultimately have $\text{eval } ?s [e] \Downarrow = \text{list-encode } (\text{rev } xs)$
by (*metis list-encode.elims diff-Suc-1 length-Cons length-greater-0-conv*)
then show $?thesis$ **using** xs * **by** *simp*
qed
qed

abbreviation $e\text{-cons} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $e\text{-cons } e \ es \equiv \text{list-encode } (e \# \text{list-decode } es)$

lemma *e-cons-elementary*:

$e\text{-cons } e \ es =$
(if $es = 0$ *then* $\text{Suc } (\text{prod-encode } (0, e))$
else $\text{Suc } (\text{prod-encode } (e\text{-length } es, \text{prod-encode } (e, \text{pdec2 } (es - 1))))$
proof (*cases* $es = 0$)
case *True*
then show $?thesis$ **by** *simp*
next
case *False*
then have $e\text{-length } es = \text{Suc } (\text{pdec1 } (es - 1))$
by (*metis list-decode.elims diff-Suc-1 tuple-decode-length*)
moreover have $es = e\text{-tl } (\text{list-encode } (e \# \text{list-decode } es))$
by (*metis list.sel(3) list-decode-encode list-encode-decode*)
ultimately show $?thesis$
using *False e-tl-elementary*
by (*metis list-decode.simps(2) diff-Suc-1 list-encode-decode prod.sel(1)*
 $\text{prod-encode-inverse snd-conv tuple-decode.simps(2)}$)
qed

definition $r\text{-cons-else} \equiv$

$Cn \ 2 \ S$
 $[Cn \ 2 \ r\text{-prod-encode}$
 $[Cn \ 2 \ r\text{-length}$
 $[Id \ 2 \ 1], Cn \ 2 \ r\text{-prod-encode } [Id \ 2 \ 0, Cn \ 2 \ r\text{-pdec2 } [Cn \ 2 \ r\text{-dec } [Id \ 2 \ 1]]]]]$

lemma *r-cons-else-prim*: $\text{prim-recfn } 2 \ r\text{-cons-else}$
unfolding $r\text{-cons-else-def}$ **by** *simp*

lemma *r-cons-else*:

$\text{eval } r\text{-cons-else } [e, es] \Downarrow =$
 $\text{Suc } (\text{prod-encode } (e\text{-length } es, \text{prod-encode } (e, \text{pdec2 } (es - 1))))$
unfolding $r\text{-cons-else-def}$ **by** *simp*

definition $r\text{-cons} \equiv$

$Cn \ 2 \ r\text{-ifz}$
 $[Id \ 2 \ 1, Cn \ 2 \ S [Cn \ 2 \ r\text{-prod-encode } [r\text{-constn } 1 \ 0, Id \ 2 \ 0]], r\text{-cons-else}]$

lemma *r-cons-prim* [*simp*]: $\text{prim-recfn } 2 \ r\text{-cons}$
unfolding $r\text{-cons-def}$ **using** $r\text{-cons-else-prim}$ **by** *simp*

lemma *r-cons* [*simp*]: $\text{eval } r\text{-cons } [e, es] \Downarrow = e\text{-cons } e \ es$
unfolding $r\text{-cons-def}$ **using** $r\text{-cons-else-prim } r\text{-cons-else } e\text{-cons-elementary}$ **by** *simp*

abbreviation $e\text{-snoc} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**

$e\text{-snoc } es \ e \equiv \text{list-encode } (\text{list-decode } es \ @ \ [e])$

lemma $e\text{-nth-snoc-small}$ [simp]:
 assumes $n < e\text{-length } b$
 shows $e\text{-nth } (e\text{-snoc } b \ z) \ n = e\text{-nth } b \ n$
 using *assms* **by** (simp add: nth-append)

lemma $e\text{-hd-snoc}$ [simp]:
 assumes $e\text{-length } b > 0$
 shows $e\text{-hd } (e\text{-snoc } b \ x) = e\text{-hd } b$

proof –
 from *assms* **have** $b \neq 0$
 using *less-imp-neq* **by** force
 then **have** $hd: e\text{-hd } b = hd \ (\text{list-decode } b)$ **by** simp
 have $e\text{-length } (e\text{-snoc } b \ x) > 0$ **by** simp
 then **have** $e\text{-snoc } b \ x \neq 0$
 using *not-gr-zero* **by** fastforce
 then **have** $e\text{-hd } (e\text{-snoc } b \ x) = hd \ (\text{list-decode } (e\text{-snoc } b \ x))$ **by** simp
 with *assms* hd **show** ?thesis **by** simp
qed

definition $r\text{-snoc} \equiv Cn \ 2 \ r\text{-rev} \ [Cn \ 2 \ r\text{-cons} \ [Id \ 2 \ 1, \ Cn \ 2 \ r\text{-rev} \ [Id \ 2 \ 0]]]$

lemma $r\text{-snoc-prim}$ [simp]: *prim-recfn* 2 $r\text{-snoc}$
 unfolding $r\text{-snoc-def}$ **by** simp

lemma $r\text{-snoc}$ [simp]: *eval* $r\text{-snoc} \ [es, \ e] \ \downarrow = e\text{-snoc } es \ e$
 unfolding $r\text{-snoc-def}$ **by** simp

abbreviation $e\text{-butlast} :: nat \Rightarrow nat$ **where**
 $e\text{-butlast } e \equiv \text{list-encode } (\text{butlast } (\text{list-decode } e))$

abbreviation $e\text{-take} :: nat \Rightarrow nat \Rightarrow nat$ **where**
 $e\text{-take } n \ x \equiv \text{list-encode } (\text{take } n \ (\text{list-decode } x))$

definition $r\text{-take} \equiv$
 $Cn \ 2 \ r\text{-ifle}$
 $[Id \ 2 \ 0, \ Cn \ 2 \ r\text{-length} \ [Id \ 2 \ 1],$
 $Pr \ 1 \ Z \ (Cn \ 3 \ r\text{-snoc} \ [Id \ 3 \ 1, \ Cn \ 3 \ r\text{-nth} \ [Id \ 3 \ 2, \ Id \ 3 \ 0]],$
 $Id \ 2 \ 1]$

lemma $r\text{-take-prim}$ [simp]: *prim-recfn* 2 $r\text{-take}$
 unfolding $r\text{-take-def}$ **by** simp-all

lemma $r\text{-take}$:
 assumes $x = \text{list-encode } es$
 shows *eval* $r\text{-take} \ [n, \ x] \ \downarrow = \text{list-encode } (\text{take } n \ es)$
proof –
 let ?g = $Cn \ 3 \ r\text{-snoc} \ [Id \ 3 \ 1, \ Cn \ 3 \ r\text{-nth} \ [Id \ 3 \ 2, \ Id \ 3 \ 0]]$
 let ?h = $Pr \ 1 \ Z \ ?g$
 have *total* ?h **using** *Mn-free-imp-total* **by** simp
 have $m \leq \text{length } es \implies \text{eval } ?h \ [m, \ x] \ \downarrow = \text{list-encode } (\text{take } m \ es)$ **for** m
 proof (*induction* m)
 case 0
 then **show** ?case **using** *assms* $r\text{-take-def}$ **by** (simp add: $r\text{-take-def}$)
 next

case (*Suc m*)
then have $m < \text{length } es$ **by** *simp*
then have $\text{eval } ?h [Suc\ m, x] = \text{eval } ?g [m, \text{the } (\text{eval } ?h [m, x]), x]$
using *Suc r-take-def* **by** *simp*
also have $\dots = \text{eval } ?g [m, \text{list-encode } (\text{take } m\ es), x]$
using *Suc* **by** *simp*
also have $\dots \Downarrow = e\text{-snoc } (\text{list-encode } (\text{take } m\ es)) (es\ !\ m)$
by (*simp add: <math>\langle m < \text{length } es \rangle\ \text{assms}</math>*)
also have $\dots \Downarrow = \text{list-encode } ((\text{take } m\ es)\ @\ [es\ !\ m])$
using *list-decode-encode* **by** *simp*
also have $\dots \Downarrow = \text{list-encode } (\text{take } (Suc\ m)\ es)$
by (*simp add: <math>\langle m < \text{length } es \rangle\ \text{take-Suc-conv-app-nth}</math>*)
finally show *?case* .
qed
moreover have $\text{eval } (Id\ 2\ 1) [m, x] \Downarrow = \text{list-encode } (\text{take } m\ es)$ **if** $m > \text{length } es$ **for** m
using *that assms* **by** *simp*
moreover have $\text{eval } r\text{-take } [m, x] \Downarrow =$
(if $m \leq e\text{-length } x$ *then the* $(\text{eval } ?h [m, x])$ *else the* $(\text{eval } (Id\ 2\ 1) [m, x])$ *)*
for m
unfolding *r-take-def* **using** *$\langle total\ ?h \rangle$* **by** *simp*
ultimately show *?thesis* **unfolding** *r-take-def* **by** *fastforce*
qed

corollary *r-take'* [*simp*]: $\text{eval } r\text{-take } [n, x] \Downarrow = e\text{-take } n\ x$
by (*simp add: r-take*)

definition *r-last* $\equiv Cn\ 1\ r\text{-hd } [r\text{-rev}]$

lemma *r-last-prim* [*simp*]: *prim-recfn 1 r-last*
unfolding *r-last-def* **by** *simp*

lemma *r-last* [*simp*]:
assumes $e = \text{list-encode } xs$ **and** $\text{length } xs > 0$
shows $\text{eval } r\text{-last } [e] \Downarrow = \text{last } xs$

proof –
from *assms(2)* **have** $\text{length } (\text{rev } xs) > 0$ **by** *simp*
then have $\text{list-encode } (\text{rev } xs) > 0$
by (*metis gr0I list.size(3) list-encode-0*)
moreover have $\text{eval } r\text{-last } [e] = \text{eval } r\text{-hd } [\text{the } (\text{eval } r\text{-rev } [e])]$
unfolding *r-last-def* **by** *simp*
ultimately show *?thesis* **using** *assms hd-rev* **by** *auto*
qed

definition *r-update-aux* \equiv
let
 $f = r\text{-constn } 2\ 0;$
 $g = Cn\ 5\ r\text{-snoc}$
 $[Id\ 5\ 1, Cn\ 5\ r\text{-ifeq } [Id\ 5\ 0, Id\ 5\ 3, Id\ 5\ 4, Cn\ 5\ r\text{-nth } [Id\ 5\ 2, Id\ 5\ 0]]]$
in Pr 3 f g

lemma *r-update-aux-recfn*: *recfn 4 r-update-aux*
unfolding *r-update-aux-def* **by** *simp*

lemma *r-update-aux*:
assumes $n \leq e\text{-length } b$
shows $\text{eval } r\text{-update-aux } [n, b, j, v] \Downarrow = \text{list-encode } ((\text{take } n\ (\text{list-decode } b))[j:=v])$

```

using assms
proof (induction n)
  case 0
    then show ?case unfolding r-update-aux-def by simp
next
  case (Suc n)
  then have n: n < e-length b
    by simp
  let ?a = Cn 5 r-nth [Id 5 2, Id 5 0]
  let ?b = Cn 5 r-ifeq [Id 5 0, Id 5 3, Id 5 4, ?a]
  define g where g ≡ Cn 5 r-snoc [Id 5 1, ?b]
  then have g: eval g [n, r, b, j, v] ↓= e-snoc r (if n = j then v else e-nth b n) for r
    by simp

  have Pr 3 (r-constn 2 0) g = r-update-aux
    using r-update-aux-def g-def by simp
  then have eval r-update-aux [Suc n, b, j, v] =
    eval g [n, the (eval r-update-aux [n, b, j, v]), b, j, v]
    using r-update-aux-recfn Suc n eval-Pr-converg-Suc
    by (metis arity.simps(5) length-Cons list.size(3) nat-less-le
      numeral-3-eq-3 option.simps(3))
  then have *: eval r-update-aux [Suc n, b, j, v] ↓= e-snoc
    (list-encode ((take n (list-decode b))[j:=v]))
    (if n = j then v else e-nth b n)
    using g Suc by simp

  consider (j-eq-n) j = n | (j-less-n) j < n | (j-gt-n) j > n
    by linarith
  then show ?case
  proof (cases)
    case j-eq-n
      moreover from this have (take (Suc n) (list-decode b))[j:=v] =
        (take n (list-decode b))[j:=v] @ [v]
        using n
        by (metis length-list-update nth-list-update-eq take-Suc-conv-app-nth take-update-swap)
        ultimately show ?thesis using * by simp
    next
      case j-less-n
      moreover from this have (take (Suc n) (list-decode b))[j:=v] =
        (take n (list-decode b))[j:=v] @ [(list-decode b) ! n]
        using n
        by (simp add: le-eq-less-or-eq list-update-append min-absorb2 take-Suc-conv-app-nth)
        ultimately show ?thesis using * by auto
    next
      case j-gt-n
      moreover from this have (take (Suc n) (list-decode b))[j:=v] =
        (take n (list-decode b))[j:=v] @ [(list-decode b) ! n]
        using n take-Suc-conv-app-nth by (auto simp: list-update-beyond)
        ultimately show ?thesis using * by auto
  qed
qed

```

abbreviation *e-update :: nat ⇒ nat ⇒ nat ⇒ nat where*
e-update b j v ≡ list-encode ((list-decode b)[j:=v])

definition *r-update ≡*

Cn 3 r-update-aux [Cn 3 r-length [Id 3 0], Id 3 0, Id 3 1, Id 3 2]

lemma *r-update-recfn [simp]: recfn 3 r-update*
unfolding *r-update-def using r-update-aux-recfn by simp*

lemma *r-update [simp]: eval r-update [b, j, v] ↓= e-update b j v*
unfolding *r-update-def using r-update-aux r-update-aux-recfn by simp*

lemma *e-length-update [simp]: e-length (e-update b k v) = e-length b*
by simp

definition *e-append :: nat ⇒ nat ⇒ nat where*
e-append xs ys ≡ list-encode (list-decode xs @ list-decode ys)

lemma *e-length-append: e-length (e-append xs ys) = e-length xs + e-length ys*
using e-append-def by simp

lemma *e-nth-append-small:*
assumes *n < e-length xs*
shows *e-nth (e-append xs ys) n = e-nth xs n*
using e-append-def assms by (simp add: nth-append)

lemma *e-nth-append-big:*
assumes *n ≥ e-length xs*
shows *e-nth (e-append xs ys) n = e-nth ys (n - e-length xs)*
using e-append-def assms e-nth by (simp add: less-diff-conv2 nth-append)

definition *r-append ≡*
let
f = Id 2 0;
g = Cn 4 r-snoc [Id 4 1, Cn 4 r-nth [Id 4 3, Id 4 0]]
in Cn 2 (Pr 2 f g) [Cn 2 r-length [Id 2 1], Id 2 0, Id 2 1]

lemma *r-append-prim [simp]: prim-recfn 2 r-append*
unfolding r-append-def by simp

lemma *r-append [simp]: eval r-append [a, b] ↓= e-append a b*

proof –

define *g where g = Cn 4 r-snoc [Id 4 1, Cn 4 r-nth [Id 4 3, Id 4 0]]*
then have *g: eval g [j, r, a, b] ↓= e-snoc r (e-nth b j) for j r*
by simp
let *?h = Pr 2 (Id 2 0) g*
have *eval ?h [n, a, b] ↓= list-encode (list-decode a @ (take n (list-decode b)))*
if *n ≤ e-length b for n*
using *that g g-def by (induction n) (simp-all add: take-Suc-conv-app-nth)*
then show *?thesis*
unfolding r-append-def g-def e-append-def by simp

qed

definition *e-append-zeros :: nat ⇒ nat ⇒ nat where*
e-append-zeros b z ≡ e-append b (list-encode (replicate z 0))

lemma *e-append-zeros-length: e-length (e-append-zeros b z) = e-length b + z*
using e-append-def e-append-zeros-def by simp

lemma *e-nth-append-zeros: e-nth (e-append-zeros b z) i = e-nth b i*

using *e-append-zeros-def e-nth-append-small e-nth-append-big* **by** *auto*

lemma *e-nth-append-zeros-big*:

assumes $i \geq e\text{-length } b$

shows $e\text{-nth } (e\text{-append-zeros } b \ z) \ i = 0$

unfolding *e-append-zeros-def*

using *e-nth-append-big[of b i list-encode (replicate z 0), OF assms(1)]*

by *simp*

definition *r-append-zeros* \equiv

r-swap (Pr 1 (Id 1 0) (Cn 3 r-snoc [Id 3 1, r-constn 2 0]))

lemma *r-append-zeros-prim* [*simp*]: *prim-recfn 2 r-append-zeros*

unfolding *r-append-zeros-def* **by** *simp*

lemma *r-append-zeros: eval r-append-zeros [b, z] \downarrow = e-append-zeros b z*

proof –

let $?r = Pr\ 1\ (Id\ 1\ 0)\ (Cn\ 3\ r\ snoc\ [Id\ 3\ 1,\ r\ constn\ 2\ 0])$

have $eval\ ?r\ [z,\ b] \ \downarrow = e\text{-append-zeros } b \ z$

using *e-append-zeros-def e-append-def*

by (*induction z*) (*simp-all add: replicate-append-same*)

then show *?thesis* **by** (*simp add: r-append-zeros-def*)

qed

end

1.5 A universal partial recursive function

theory *Universal*

imports *Partial-Recursive*

begin

The main product of this section is a universal partial recursive function, which given a code i of an n -ary partial recursive function f and an encoded list xs of n arguments, computes $eval\ f\ xs$. From this we can derive fixed-arity universal functions satisfying the usual results such as the s - m - n theorem. To represent the code i , we need a way to encode *recfs* as natural numbers (Section 1.5.2). To construct the universal function, we devise a ternary function taking i , xs , and a step bound t and simulating the execution of f on input xs for t steps. This function is useful in its own right, enabling techniques like dovetailing or “concurrent” evaluation of partial recursive functions.

The notion of a “step” is not part of the definition of (the evaluation of) partial recursive functions, but one can simulate the evaluation on an abstract machine (Section 1.5.1). This machine’s configurations can be encoded as natural numbers, and this leads us to a step function $nat \Rightarrow nat$ on encoded configurations (Section 1.5.3). This function in turn can be computed by a primitive recursive function, from which we develop the aforementioned ternary function of i , xs , and t (Section 1.5.4). From this we can finally derive a universal function (Section 1.5.5).

1.5.1 A step function

We simulate the stepwise execution of a partial recursive function in a fairly straightforward way reminiscent of the execution of function calls in an imperative programming language. A configuration of the abstract machine is a pair consisting of:

1. A stack of frames. A frame represents the execution of a function and is a triple $(f, xs, locals)$ of
 - (a) a *recf* f being executed,
 - (b) a *nat list* of arguments of f ,
 - (c) a *nat list* of local variables, which holds intermediate values when f is of the form Cn , Pr , or Mn .
2. A register of type *nat option* representing the return value of the last function call: *None* signals that in the previous step the stack was not popped and hence no value was returned, whereas *Some v* means that in the previous step a function returned v .

For computing h on input xs , the initial configuration is $([(h, xs, [])], None)$. When the computation for a frame ends, it is popped off the stack, and its return value is put in the register. The entire computation ends when the stack is empty. In such a final configuration the register contains the value of h at xs . If no final configuration is ever reached, h diverges at xs .

The execution of one step depends on the topmost (that is, active) frame. In the step when a frame $(h, xs, locals)$ is pushed onto the stack, the local variables are $locals = []$. The following happens until the frame is popped off the stack again (if it ever is):

- For the base functions $h = Z$, $h = S$, $h = Id\ m\ n$, the frame is popped off the stack right away, and the return value is placed in the register.
- For $h = Cn\ n\ f\ gs$, for each function g in gs :
 1. A new frame of the form $(g, xs, [])$ is pushed onto the stack.
 2. When (and if) this frame is eventually popped, the value in the register is $eval\ g\ xs$. This value is appended to the list $locals$ of local variables.

When all g in gs have been evaluated in this manner, f is evaluated on the local variables by pushing $(f, locals, [])$. The resulting register value is kept and the active frame for h is popped off the stack.

- For $h = Pr\ n\ f\ g$, let $xs = y \# ys$. First $(f, ys, [])$ is pushed and the return value stored in the $locals$. Then $(g, x \# v \# ys, [])$ is pushed, where x is the length of $locals$ and v the most recently appended value. The return value is appended to $locals$. This is repeated until the length of $locals$ reaches y . Then the most recently appended local is placed in the register, and the stack is popped.
- For $h = Mn\ n\ f$, frames $(f, x \# xs, [])$ are pushed for $x = 0, 1, 2, \dots$ until one of them returns 0. Then this x is placed in the register and the stack is popped. Until then x is stored in $locals$. If none of these evaluations return 0, the stack never shrinks, and thus the machine never reaches a final state.

type-synonym $frame = recf \times nat\ list \times nat\ list$

type-synonym $configuration = frame\ list \times nat\ option$

Definition of the step function

```

fun step :: configuration ⇒ configuration where
  step ([], rv) = ([], rv)
| step (((Z, -, -) # fs), rv) = (fs, Some 0)
| step (((S, xs, -) # fs), rv) = (fs, Some (Suc (hd xs)))
| step (((Id m n, xs, -) # fs), rv) = (fs, Some (xs ! n))
| step (((Cn n f gs, xs, ls) # fs), rv) =
  (if length ls = length gs
   then if rv = None
        then ((f, ls, []) # (Cn n f gs, xs, ls) # fs, None)
        else (fs, rv)
   else if rv = None
        then if length ls < length gs
              then ((gs ! (length ls), xs, []) # (Cn n f gs, xs, ls) # fs, None)
              else (fs, rv) — cannot occur, so don't-care term
        else ((Cn n f gs, xs, ls @ [the rv]) # fs, None))
| step (((Pr n f g, xs, ls) # fs), rv) =
  (if ls = []
   then if rv = None
        then ((f, tl xs, []) # (Pr n f g, xs, ls) # fs, None)
        else ((Pr n f g, xs, [the rv]) # fs, None)
   else if length ls = Suc (hd xs)
        then (fs, Some (hd ls))
        else if rv = None
              then ((g, (length ls - 1) # hd ls # tl xs, []) # (Pr n f g, xs, ls) # fs, None)
              else ((Pr n f g, xs, (the rv) # ls) # fs, None))
| step (((Mn n f, xs, ls) # fs), rv) =
  (if ls = []
   then ((f, 0 # xs, []) # (Mn n f, xs, [0]) # fs, None)
   else if rv = Some 0
        then (fs, Some (hd ls))
        else ((f, (Suc (hd ls)) # xs, []) # (Mn n f, xs, [Suc (hd ls)]) # fs, None))

```

definition reachable :: configuration ⇒ configuration ⇒ bool **where**
 reachable x y ≡ ∃ t. iterate t step x = y

lemma step-reachable [intro]:

assumes step x = y
shows reachable x y
unfolding reachable-def **using** assms **by** (metis iterate.simps(1,2) comp-id)

lemma reachable-transitive [trans]:

assumes reachable x y **and** reachable y z
shows reachable x z
using assms iterate-additive[**where** ?f=step] reachable-def **by** metis

lemma reachable-refl: reachable x x

unfolding reachable-def **by** (metis iterate.simps(1) eq-id-iff)

From a final configuration, that is, when the stack is empty, only final configurations are reachable.

lemma step-empty-stack:

assumes fst x = []
shows fst (step x) = []
using assms **by** (metis prod.collapse step.simps(1))

lemma *reachable-empty-stack*:
assumes $\text{fst } x = []$ **and** *reachable* $x\ y$
shows $\text{fst } y = []$
proof –
have $\text{fst } (\text{iterate } t \text{ step } x) = []$ **for** t
using *assms step-empty-stack* **by** (*induction t simp-all*)
then show *?thesis*
using *reachable-def assms(2)* **by** *auto*
qed

abbreviation *nonterminating* $:: \text{configuration} \Rightarrow \text{bool}$ **where**
nonterminating $x \equiv \forall t. \text{fst } (\text{iterate } t \text{ step } x) \neq []$

lemma *reachable-nonterminating*:
assumes *reachable* $x\ y$ **and** *nonterminating* y
shows *nonterminating* x
proof –
from *assms(1)* **obtain** t_1 **where** $t1: \text{iterate } t_1 \text{ step } x = y$
using *reachable-def* **by** *auto*
have $\text{fst } (\text{iterate } t \text{ step } x) \neq []$ **for** t
proof (*cases* $t \leq t_1$)
case *True*
then show *?thesis*
using $t1$ *assms(2) reachable-def reachable-empty-stack iterate-additive'*
by (*metis le-Suc-ex*)
next
case *False*
then have $\text{iterate } t \text{ step } x = \text{iterate } (t_1 + (t - t_1)) \text{ step } x$
by *simp*
then have $\text{iterate } t \text{ step } x = \text{iterate } (t - t_1) \text{ step } (\text{iterate } t_1 \text{ step } x)$
by (*simp add: iterate-additive'*)
then have $\text{iterate } t \text{ step } x = \text{iterate } (t - t_1) \text{ step } y$
using $t1$ **by** *simp*
then show $\text{fst } (\text{iterate } t \text{ step } x) \neq []$
using *assms(2)* **by** *simp*
qed
then show *?thesis ..*
qed

The function *step* is underdefined, for example, when the top frame contains a non-well-formed *recf* or too few arguments. All is well, though, if every frame contains a well-formed *recf* whose arity matches the number of arguments. Such stacks will be called *valid*.

definition *valid* $:: \text{frame list} \Rightarrow \text{bool}$ **where**
valid $\text{stack} \equiv \forall s \in \text{set } \text{stack}. \text{recfn } (\text{length } (\text{fst } (\text{snd } s))) (\text{fst } s)$

lemma *valid-frame*: $\text{valid } (s \# \text{ss}) \Longrightarrow \text{valid } \text{ss} \wedge \text{recfn } (\text{length } (\text{fst } (\text{snd } s))) (\text{fst } s)$
using *valid-def* **by** *simp*

lemma *valid-ConsE*: $\text{valid } ((f, \text{xs}, \text{locs}) \# \text{rest}) \Longrightarrow \text{valid } \text{rest} \wedge \text{recfn } (\text{length } \text{xs}) f$
using *valid-def* **by** *simp*

lemma *valid-ConsI*: $\text{valid } \text{rest} \Longrightarrow \text{recfn } (\text{length } \text{xs}) f \Longrightarrow \text{valid } ((f, \text{xs}, \text{locs}) \# \text{rest})$
using *valid-def* **by** *simp*

Stacks in initial configurations are valid, and performing a step maintains the validity of the stack.

```

lemma step-valid: valid stack  $\implies$  valid (fst (step (stack, rv)))
proof (cases stack)
  case Nil
  then show ?thesis using valid-def by simp
next
  case (Cons s ss)
  assume valid: valid stack
  then have *: valid ss  $\wedge$  recfn (length (fst (snd s))) (fst s)
    using valid-frame Cons by simp
  show ?thesis
  proof (cases fst s)
    case Z
    then show ?thesis using Cons valid * by (metis fstI prod.collapse step.simps(2))
  next
    case S
    then show ?thesis using Cons valid * by (metis fst-conv prod.collapse step.simps(3))
  next
    case Id
    then show ?thesis using Cons valid * by (metis fstI prod.collapse step.simps(4))
  next
    case (Cn n f gs)
    then obtain xs ls where s = (Cn n f gs, xs, ls)
      using Cons by (metis prod.collapse)
    moreover consider
      length ls = length gs  $\wedge$  rv  $\uparrow$ 
    | length ls = length gs  $\wedge$  rv  $\downarrow$ 
    | length ls < length gs  $\wedge$  rv  $\uparrow$ 
    | length ls  $\neq$  length gs  $\wedge$  rv  $\downarrow$ 
    | length ls > length gs  $\wedge$  rv  $\uparrow$ 
      by linarith
    ultimately show ?thesis using valid Cons valid-def by (cases) auto
  next
    case (Pr n f g)
    then obtain xs ls where s: s = (Pr n f g, xs, ls)
      using Cons by (metis prod.collapse)
    consider
      length ls = 0  $\wedge$  rv  $\uparrow$ 
    | length ls = 0  $\wedge$  rv  $\downarrow$ 
    | length ls  $\neq$  0  $\wedge$  length ls = Suc (hd xs)
    | length ls  $\neq$  0  $\wedge$  length ls  $\neq$  Suc (hd xs)  $\wedge$  rv  $\uparrow$ 
    | length ls  $\neq$  0  $\wedge$  length ls  $\neq$  Suc (hd xs)  $\wedge$  rv  $\downarrow$ 
      by linarith
    then show ?thesis using Cons * valid-def s by (cases) auto
  next
    case (Mn n f)
    then obtain xs ls where s: s = (Mn n f, xs, ls)
      using Cons by (metis prod.collapse)
    consider
      length ls = 0
    | length ls  $\neq$  0  $\wedge$  rv  $\uparrow$ 
    | length ls  $\neq$  0  $\wedge$  rv  $\downarrow$ 
      by linarith
    then show ?thesis using Cons * valid-def s by (cases) auto

```

qed
qed

corollary *iterate-step-valid*:

assumes *valid stack*

shows *valid (fst (iterate t step (stack, rv)))*

using *assms*

proof (*induction t*)

case *0*

then show *?case* **by** *simp*

next

case (*Suc t*)

moreover have *iterate (Suc t) step (stack, rv) = step (iterate t step (stack, rv))*

by *simp*

ultimately show *?case* **using** *step-valid valid-def* **by** (*metis prod.collapse*)

qed

Correctness of the step function

The function *step* works correctly for a *recf f* on arguments *xs* in some configuration if (1) in case *f* converges, *step* reaches a configuration with the topmost frame popped and *eval f xs* in the register, and (2) in case *f* diverges, *step* does not reach a final configuration.

fun *correct* :: *configuration* \Rightarrow *bool* **where**

correct (\square , *r*) = *True*

| *correct* ((*f*, *xs*, *ls*) # *rest*, *r*) =

(if *eval f xs* \downarrow then *reachable* ((*f*, *xs*, *ls*) # *rest*, *r*) (*rest*, *eval f xs*)

else *nonterminating* ((*f*, *xs*, *ls*) # *rest*, *None*))

lemma *correct-convergI*:

assumes *eval f xs* \downarrow **and** *reachable* ((*f*, *xs*, *ls*) # *rest*, *None*) (*rest*, *eval f xs*)

shows *correct* ((*f*, *xs*, *ls*) # *rest*, *None*)

using *assms* **by** *auto*

lemma *correct-convergE*:

assumes *correct* ((*f*, *xs*, *ls*) # *rest*, *None*) **and** *eval f xs* \downarrow

shows *reachable* ((*f*, *xs*, *ls*) # *rest*, *None*) (*rest*, *eval f xs*)

using *assms* **by** *simp*

The correctness proof for *step* is by structural induction on the *recf* in the top frame. The base cases *Z*, *S*, and *Id* are simple. For $X = Cn, Pr, Mn$, the lemmas named *reachable-X* show which configurations are reachable for *recfs* of shape *X*. Building on those, the lemmas named *step-X-correct* show *step*'s correctness for *X*.

lemma *reachable-Cn*:

assumes *valid* (((*Cn n f gs*), *xs*, \square) # *rest*) (**is valid** *?stack*)

and $\bigwedge xs$ *rest*. *valid* ((*f*, *xs*, \square) # *rest*) \implies *correct* ((*f*, *xs*, \square) # *rest*, *None*)

and $\bigwedge g$ *xs* *rest*.

$g \in \text{set } gs \implies \text{valid} ((g, xs, \square) \# \text{rest}) \implies \text{correct} ((g, xs, \square) \# \text{rest}, \text{None})$

and $\forall i < k$. *eval* (*gs* ! *i*) *xs* \downarrow

and $k \leq \text{length } gs$

shows *reachable*

(*?stack*, *None*)

((*Cn n f gs*, *xs*, *take k* (*map* (λg . *the* (*eval g xs*)) *gs*)) # *rest*, *None*)

using *assms*(4,5)

```

proof (induction k)
  case 0
  then show ?case using reachable-refl by simp
next
  case (Suc k)
  let ?ys = map (λg. the (eval g xs)) gs
  from Suc have k < length gs by simp
  have valid: recfn (length xs) (Cn n f gs) valid rest
    using assms(1) valid-ConsE[of (Cn n f gs)] by simp-all
  from Suc have reachable (?stack, None) ((Cn n f gs, xs, take k ?ys) # rest, None)
    (is - (?stack1, None))
    by simp
  also have reachable ... ((gs ! k, xs, []) # ?stack1, None)
    using step-reachable ⟨k < length gs⟩
    by (auto simp: min-absorb2)
  also have reachable ... (?stack1, eval (gs ! k) xs)
    (is - (-, ?rv))
    using Suc.premis(1) ⟨k < length gs⟩ assms(3) valid valid-ConsI by auto
  also have reachable ... ((Cn n f gs, xs, (take (Suc k) ?ys)) # rest, None)
    (is - (?stack2, None))
  proof -
    have step (?stack1, ?rv) = ((Cn n f gs, xs, (take k ?ys) @ [the ?rv]) # rest, None)
      using Suc by auto
    also have ... = ((Cn n f gs, xs, (take (Suc k) ?ys)) # rest, None)
      by (simp add: ⟨k < length gs⟩ take-Suc-conv-app-nth)
    finally show ?thesis
      using step-reachable by auto
  qed
  finally show reachable (?stack, None) (?stack2, None) .
qed

```

lemma step-Cn-correct:

```

assumes valid (((Cn n f gs), xs, []) # rest) (is valid ?stack)
  and ∧xs rest. valid ((f, xs, []) # rest) ⇒ correct ((f, xs, []) # rest, None)
  and ∧g xs rest.
    g ∈ set gs ⇒ valid ((g, xs, []) # rest) ⇒ correct ((g, xs, []) # rest, None)
shows correct (?stack, None)

```

proof -

```

have valid: recfn (length xs) (Cn n f gs) valid rest
  using valid-ConsE[OF assms(1)] by auto
let ?ys = map (λg. the (eval g xs)) gs
consider
  (diverg-f) ∨ g ∈ set gs. eval g xs ↓ and eval f ?ys ↑
  | (diverg-gs) ∃ g ∈ set gs. eval g xs ↑
  | (converg) eval (Cn n f gs) xs ↓
  using valid-ConsE[OF assms(1)] by fastforce
then show ?thesis
proof (cases)
  case diverg-f
  then have ∨ i < length gs. eval (gs ! i) xs ↓ by simp
  then have reachable (?stack, None) ((Cn n f gs, xs, ?ys) # rest, None)
    (is - (?stack1, None))
    using reachable-Cn[OF assms, where ?k=length gs] by simp
  also have reachable ... ((f, ?ys, []) # ?stack1, None) (is - (?stack2, None))
    by (simp add: step-reachable)
  finally have reachable (?stack, None) (?stack2, None) .

```

moreover have *nonterminating* (?stack2, None)
using *diverg-f(2) assms(2)[of ?ys ?stack1] valid-ConsE[OF assms(1)] valid-ConsI*
by *auto*
ultimately have *nonterminating* (?stack, None)
using *reachable-nonterminating* **by** *simp*
moreover have *eval (Cn n f gs) xs* \uparrow
using *diverg-f(2) assms(1) eval-Cn valid-ConsE* **by** *presburger*
ultimately show *?thesis* **by** *simp*
next
case *diverg-gs*
then have *ex-i: $\exists i < \text{length } gs. \text{eval } (gs ! i) \text{ xs}$* \uparrow
using *in-set-conv-nth[of - gs]* **by** *auto*
define *k* **where** *k = (LEAST i. i < length gs \wedge eval (gs ! i) xs \uparrow)* (**is** - = *Least ?P*)
then have *gs-k: eval (gs ! k) xs* \uparrow
using *LeastI-ex[OF ex-i]* **by** *simp*
have $\forall i < k. \text{eval } (gs ! i) \text{ xs}$ \downarrow
using *k-def not-less-Least[of - ?P] LeastI-ex[OF ex-i]* **by** *simp*
moreover from this have *k < length gs*
using *ex-i less-le-trans not-le* **by** *blast*
ultimately have *reachable* (?stack, None) ((*Cn n f gs, xs, take k ?ys*) # *rest, None*)
using *reachable-Cn[OF assms]* **by** *simp*
also have *reachable ...*
(*(gs ! (length (take k ?ys)), xs, [])* # (*Cn n f gs, xs, take k ?ys*) # *rest, None*)
(**is** - (*?stack1, None*))
proof -
have *length (take k ?ys) < length gs*
by (*simp add: <k < length gs> less-imp-le-nat min-less-iff-disj*)
then show *?thesis* **using** *step-reachable <k < length gs>*
by *auto*
qed
finally have *reachable* (?stack, None) (?stack1, None) .
moreover have *nonterminating* (?stack1, None)
proof -
have *recfn (length xs) (gs ! k)*
using *<k < length gs> valid(1)* **by** *simp*
then have *correct* (?stack1, None)
using *<k < length gs> nth-mem valid valid-ConsI*
assms(3)[of gs ! (length (take k ?ys)) xs]
by *auto*
moreover have *length (take k ?ys) = k*
by (*simp add: <k < length gs> less-imp-le-nat min-absorb2*)
ultimately show *?thesis* **using** *gs-k* **by** *simp*
qed
ultimately have *nonterminating* (?stack, None)
using *reachable-nonterminating* **by** *simp*
moreover have *eval (Cn n f gs) xs* \uparrow
using *diverg-gs valid* **by** *fastforce*
ultimately show *?thesis* **by** *simp*
next
case *converg*
then have *f: eval f ?ys* \downarrow **and** *g: $\bigwedge g. g \in \text{set } gs \implies \text{eval } g \text{ xs}$* \downarrow
using *valid(1)* **by** (*metis eval-Cn*)
then have $\forall i < \text{length } gs. \text{eval } (gs ! i) \text{ xs}$ \downarrow
by *simp*
then have *reachable* (?stack, None) ((*Cn n f gs, xs, take (length gs) ?ys*) # *rest, None*)
using *reachable-Cn assms* **by** *blast*

```

also have reachable ... ((Cn n f gs, xs, ?ys) # rest, None) (is - (?stack1, None))
  by (simp add: reachable-refl)
also have reachable ... ((f, ?ys, []) # ?stack1, None)
  using step-reachable by auto
also have reachable ... (?stack1, eval f ?ys)
  using assms(2)[of ?ys] correct-convergE valid f valid-ConsI by auto
also have reachable (?stack1, eval f ?ys) (rest, eval f ?ys)
  using f by auto
finally have reachable (?stack, None) (rest, eval f ?ys) .
moreover have eval (Cn n f gs) xs = eval f ?ys
  using g valid(1) by auto
ultimately show ?thesis
  using converg correct-convergI by auto
qed
qed

During the execution of a frame with a partial recursive function of shape  $Pr\ n\ f\ g$  and
arguments  $x\ \#\ xs$ , the list of local variables collects all the function values up to  $x$ 
in reversed order. We call such a list a trace for short.

definition trace :: nat  $\Rightarrow$  recf  $\Rightarrow$  recf  $\Rightarrow$  nat list  $\Rightarrow$  nat  $\Rightarrow$  nat list where
  trace n f g xs x  $\equiv$  map ( $\lambda y$ . the (eval (Pr n f g) (y # xs))) (rev [0.. $Suc\ x$ ])

lemma trace-length: length (trace n f g xs x) = Suc x
  using trace-def by simp

lemma trace-hd: hd (trace n f g xs x) = the (eval (Pr n f g) (x # xs))
  using trace-def by simp

lemma trace-Suc:
  trace n f g xs (Suc x) = (the (eval (Pr n f g) (Suc x # xs))) # (trace n f g xs x)
  using trace-def by simp

lemma reachable-Pr:
  assumes valid (((Pr n f g), x # xs, []) # rest) (is valid ?stack)
  and  $\bigwedge xs\ rest$ . valid ((f, xs, []) # rest)  $\implies$  correct ((f, xs, []) # rest, None)
  and  $\bigwedge xs\ rest$ . valid ((g, xs, []) # rest)  $\implies$  correct ((g, xs, []) # rest, None)
  and  $y \leq x$ 
  and eval (Pr n f g) (y # xs)  $\downarrow$ 
shows reachable (?stack, None) ((Pr n f g, x # xs, trace n f g xs y) # rest, None)
  using assms(4,5)
proof (induction y)
  case 0
  have valid: recfn (length (x # xs)) (Pr n f g) valid rest
    using valid-ConsE[OF assms(1)] by simp-all
  then have f: eval f xs  $\downarrow$  using 0 by simp
  let ?as = x # xs
  have reachable (?stack, None) ((f, xs, []) # ((Pr n f g), ?as, []) # rest, None)
    using step-reachable by auto
  also have reachable ... (?stack, eval f xs)
    using assms(2)[of xs ((Pr n f g), ?as, []) # rest]
      correct-convergE[OF - f] f valid valid-ConsI
    by simp
  also have reachable ... ((Pr n f g, ?as, [the (eval f xs)]) # rest, None)
    using step-reachable valid(1) f by auto
  finally have reachable (?stack, None) ((Pr n f g, ?as, [the (eval f xs)]) # rest, None) .
  then show ?case using trace-def valid(1) by simp

```

next
case (*Suc y*)
have *valid*: *recfn* (*length* (*x # xs*)) (*Pr n f g*) *valid rest*
using *valid-ConsE*[*OF assms*(1)] **by** *simp-all*
let *?ls = trace n f g xs y*
have *lenls*: *length ?ls = Suc y*
using *trace-length* **by** *auto*
moreover have *hdls*: *hd ?ls = the (eval (Pr n f g) (y # xs))*
using *Suc trace-hd* **by** *auto*
ultimately have *g*:
eval g (y # hd ?ls # xs) ↓
eval (Pr n f g) (Suc y # xs) = eval g (y # hd ?ls # xs)
using *eval-Pr-Suc-converg hdls valid*(1) *Suc* **by** *simp-all*
then have *reachable* (*?stack*, *None*) ((*Pr n f g*, *x # xs*, *?ls*) # *rest*, *None*)
(*is -* (*?stack1*, *None*))
using *Suc valid*(1) **by** *fastforce*
also have *reachable* ... ((*g*, *y # hd ?ls # xs*, []) # (*Pr n f g*, *x # xs*, *?ls*) # *rest*, *None*)
using *Suc.prem*s *lenls* **by** *fastforce*
also have *reachable* ... (*?stack1*, *eval g (y # hd ?ls # xs)*)
(*is -* (*-*, *?rv*))
using *assms*(3) *g*(1) *valid valid-ConsI* **by** *auto*
also have *reachable* ... ((*Pr n f g*, *x # xs*, (*the ?rv*) # *?ls*) # *rest*, *None*)
using *Suc.prem*s(1) *g*(1) *lenls* **by** *auto*
finally have *reachable* (*?stack*, *None*) ((*Pr n f g*, *x # xs*, (*the ?rv*) # *?ls*) # *rest*, *None*) .
moreover have *trace n f g xs (Suc y) = (the ?rv) # ?ls*
using *g*(2) *trace-Suc* **by** *simp*
ultimately show *?case* **by** *simp*
qed

lemma *step-Pr-correct*:

assumes *valid* (((*Pr n f g*), *xs*, []) # *rest*) (**is** *valid ?stack*)
and $\bigwedge xs \text{ rest. } \text{valid} ((f, xs, []) \# \text{rest}) \implies \text{correct} ((f, xs, []) \# \text{rest}, \text{None})$
and $\bigwedge xs \text{ rest. } \text{valid} ((g, xs, []) \# \text{rest}) \implies \text{correct} ((g, xs, []) \# \text{rest}, \text{None})$
shows *correct* (*?stack*, *None*)

proof –

have *valid*: *valid rest recfn* (*length xs*) (*Pr n f g*)
using *valid-ConsE*[*OF assms*(1)] **by** *simp-all*
then have *length xs > 0*
by *auto*

then obtain *y ys* **where** *y-ys*: *xs = y # ys*
using *list.exhaust-sel* **by** *auto*

let *?t = trace n f g ys*

consider

(*converg*) *eval (Pr n f g) xs ↓*
| (*diverg-f*) *eval (Pr n f g) xs ↑* **and** *eval f ys ↑*
| (*diverg*) *eval (Pr n f g) xs ↑* **and** *eval f ys ↓*
by *auto*

then show *?thesis*

proof (*cases*)

case *converg*

then have $\bigwedge z. z \leq y \implies \text{reachable} (\text{?stack}, \text{None}) (((Pr n f g), xs, ?t z) \# \text{rest}, \text{None})$
using *assms valid* **by** (*simp add*: *eval-Pr-converg-le reachable-Pr y-ys*)

then have *reachable* (*?stack*, *None*) (((*Pr n f g*), *xs*, *?t y*) # *rest*, *None*)
by *simp*

moreover have *reachable* (((*Pr n f g*), *xs*, *?t y*) # *rest*, *None*) (*rest*, *Some (hd (?t y))*)
using *trace-length step-reachable y-ys* **by** *fastforce*

```

ultimately have reachable (?stack, None) (rest, Some (hd (?t y)))
  using reachable-transitive by blast
then show ?thesis
  using assms(1) trace-hd converg y-ys by simp
next
case diverg-f
have *: step (?stack, None) = ((f, ys, []) # ((Pr n f g), xs, []) # tl ?stack, None)
  (is - = (?stack1, None))
  using assms(1,2) y-ys by simp
then have reachable (?stack, None) (?stack1, None)
  using step-reachable by force
moreover have nonterminating (?stack1, None)
  using assms diverg-f valid valid-ConsI * by auto
ultimately have nonterminating (?stack, None)
  using reachable-nonterminating by blast
then show ?thesis using diverg-f(1) assms(1) by simp
next
case diverg
let ?h = λz. the (eval (Pr n f g) (z # ys))
let ?Q = λz. z < y ∧ eval (Pr n f g) (z # ys) ↓
have ?Q 0
  using assms diverg neq0-conv y-ys valid by fastforce
define zmax where zmax = Greatest ?Q
then have ?Q zmax
  using ⟨?Q 0⟩ GreatestI-nat[of ?Q 0 y] by simp
have le-zmax: ∧z. ?Q z ⇒ z ≤ zmax
  using Greatest-le-nat[of ?Q - y] zmax-def by simp
have len: length (?t zmax) < Suc y
  by (simp add: ⟨?Q zmax⟩ trace-length)
have eval (Pr n f g) (y # ys) ↓ if y ≤ zmax for y
  using that zmax-def ⟨?Q zmax⟩ assms eval-Pr-converg-le[of n f g ys zmax y] valid y-ys
  by simp
then have reachable (?stack, None) (((Pr n f g), xs, ?t y) # rest, None)
  if y ≤ zmax for y
  using that ⟨?Q zmax⟩ diverg y-ys assms reachable-Pr by simp
then have reachable (?stack, None) (((Pr n f g), xs, ?t zmax) # rest, None)
  (is reachable - (?stack1, None))
  by simp
also have reachable ...
  ((g, zmax # ?h zmax # tl xs, []) # (Pr n f g, xs, ?t zmax) # rest, None)
  (is - (?stack2, None))
proof (rule step-reachable)
have length (?t zmax) ≠ Suc (hd xs)
  using len y-ys by simp
moreover have hd (?t zmax) = ?h zmax
  using trace-hd by auto
moreover have length (?t zmax) = Suc zmax
  using trace-length by simp
ultimately show step (?stack1, None) = (?stack2, None)
  by auto
qed
finally have reachable (?stack, None) (?stack2, None) .
moreover have nonterminating (?stack2, None)
proof -
have correct (?stack2, None)
  using y-ys assms valid-ConsI valid by simp

```

```

moreover have eval g (zmax # ?h zmax # ys) ↑
  using ⟨?Q zmax⟩ diverg le-zmax len less-Suc-eq trace-length y-ys valid
  by fastforce
ultimately show ?thesis using y-ys by simp
qed
ultimately have nonterminating (?stack, None)
  using reachable-nonterminating by simp
then show ?thesis using diverg assms(1) by simp
qed
qed

```

lemma *reachable-Mn*:

```

assumes valid ((Mn n f, xs, []) # rest) (is valid ?stack)
  and  $\bigwedge xs \text{ rest. valid } ((f, xs, []) \# \text{rest}) \implies \text{correct } ((f, xs, []) \# \text{rest}, \text{None})$ 
  and  $\forall y < z. \text{eval } f (y \# xs) \notin \{\text{None}, \text{Some } 0\}$ 
shows reachable (?stack, None) ((f, z # xs, []) # (Mn n f, xs, [z]) # rest, None)
using assms(3)
proof (induction z)
  case 0
  then have step (?stack, None) = ((f, 0 # xs, []) # (Mn n f, xs, [0]) # rest, None)
    using assms by simp
  then show ?case
    using step-reachable assms(1) by force
next
  case (Suc z)
  have valid: valid rest recfn (length xs) (Mn n f)
    using valid-ConsE[OF assms(1)] by auto
  have f: eval f (z # xs)  $\notin \{\text{None}, \text{Some } 0\}$ 
    using Suc by simp
  have reachable (?stack, None) ((f, z # xs, []) # (Mn n f, xs, [z]) # rest, None)
    using Suc by simp
  also have reachable ... ((Mn n f, xs, [z]) # rest, eval f (z # xs))
    using f assms(2)[of z # xs] valid correct-convergE valid-ConsI by auto
  also have reachable ... ((f, (Suc z) # xs, []) # (Mn n f, xs, [Suc z]) # rest, None)
    (is - (?stack1, None))
    using step-reachable f by force
  finally have reachable (?stack, None) (?stack1, None) .
  then show ?case by simp
qed

```

lemma *iterate-step-empty-stack*: $\text{iterate } t \text{ step } ([], rv) = ([], rv)$

using step-empty-stack **by** (induction t) simp-all

lemma *reachable-iterate-step-empty-stack*:

```

assumes reachable cfg ([], rv)
shows  $\exists t. \text{iterate } t \text{ step } \text{cfg} = ([], rv) \wedge (\forall t' < t. \text{fst } (\text{iterate } t' \text{ step } \text{cfg}) \neq [])$ 
proof -
  let ?P =  $\lambda t. \text{iterate } t \text{ step } \text{cfg} = ([], rv)$ 
  from assms have  $\exists t. ?P t$ 
    by (simp add: reachable-def)
  moreover define tmin where tmin = Least ?P
  ultimately have ?P tmin
    using LeastI-ex[of ?P] by simp
  have fst (iterate t' step cfg)  $\neq []$  if t' < tmin for t'
  proof
    assume fst (iterate t' step cfg) = []

```

```

then obtain  $v$  where  $v$ : iterate  $t'$  step  $cfg = ([], v)$ 
  by (metis prod.exhaust-sel)
then have iterate  $t''$  step  $([], v) = ([], v)$  for  $t''$ 
  using iterate-step-empty-stack by simp
then have iterate  $(t' + t'')$  step  $cfg = ([], v)$  for  $t''$ 
  using  $v$  iterate-additive by fast
moreover obtain  $t''$  where  $t' + t'' = tmin$ 
  using  $\langle t' < tmin \rangle$  less-imp-add-positive by auto
ultimately have iterate  $tmin$  step  $cfg = ([], v)$ 
  by auto
then have  $v = rv$ 
  using  $\langle ?P\ tmin \rangle$  by simp
then have iterate  $t'$  step  $cfg = ([], rv)$ 
  using  $v$  by simp
moreover have  $\forall t' < tmin. \neg ?P\ t'$ 
  unfolding tmin-def using not-less-Least[of - ?P] by simp
ultimately show False
  using that by simp
qed
then show ?thesis using  $\langle ?P\ tmin \rangle$  by auto
qed

lemma step-Mn-correct:
  assumes valid  $((Mn\ n\ f,\ xs,\ []) \# rest)$  (is valid ?stack)
  and  $\bigwedge xs\ rest. \text{valid } ((f,\ xs,\ []) \# rest) \implies \text{correct } ((f,\ xs,\ []) \# rest,\ None)$ 
  shows correct  $(?stack,\ None)$ 
proof –
  have valid: valid rest recfn  $(length\ xs)\ (Mn\ n\ f)$ 
    using valid-ConsE[OF assms(1)] by auto
  consider
     $(diverg)\ \text{eval } (Mn\ n\ f)\ xs \uparrow$  and  $\forall z. \text{eval } f\ (z \# xs) \downarrow$ 
  |  $(diverg-f)\ \text{eval } (Mn\ n\ f)\ xs \uparrow$  and  $\exists z. \text{eval } f\ (z \# xs) \uparrow$ 
  |  $(converg)\ \text{eval } (Mn\ n\ f)\ xs \downarrow$ 
  by fast
  then show ?thesis
proof (cases)
  case diverg
  then have  $\forall z. \text{eval } f\ (z \# xs) \neq \text{Some } 0$ 
    using eval-Mn-diverg[OF valid(2)] by simp
  then have  $\forall y < z. \text{eval } f\ (y \# xs) \notin \{None,\ \text{Some } 0\}$  for  $z$ 
    using diverg by simp
  then have reach-z:
     $\bigwedge z. \text{reachable } (?stack,\ None)\ ((f,\ z \# xs,\ []) \# (Mn\ n\ f,\ xs,\ [z]) \# rest,\ None)$ 
    using reachable-Mn[OF assms] diverg by simp

  define  $h :: nat \Rightarrow \text{configuration}$  where
     $h\ z \equiv ((f,\ z \# xs,\ []) \# (Mn\ n\ f,\ xs,\ [z]) \# rest,\ None)$  for  $z$ 
  then have h-inj:  $\bigwedge x\ y. x \neq y \implies h\ x \neq h\ y$  and z-neq-Nil:  $\bigwedge z. \text{fst } (h\ z) \neq []$ 
    by simp-all

  have  $z: \exists z_0. \forall z > z_0. \neg (\exists t' \leq t. \text{iterate } t'\ \text{step } (?stack,\ None) = h\ z)$  for  $t$ 
  proof (induction t)
  case  $0$ 
  then show ?case by (metis h-inj le-zero-eq less-not-refl3)
  next
  case (Suc t)

```

```

then show ?case
  using h-inj by (metis (no-types, opaque-lifting) le-Suc-eq less-not-refl3 less-trans)
qed

have nonterminating (?stack, None)
proof (rule ccontr)
  assume  $\neg$  nonterminating (?stack, None)
  then obtain t where t: fst (iterate t step (?stack, None)) = []
    by auto
  then obtain z0 where  $\forall z > z_0. \neg (\exists t' \leq t. \text{iterate } t' \text{ step } (?stack, None) = h z)$ 
    using z by auto
  then have not-h:  $\forall t' \leq t. \text{iterate } t' \text{ step } (?stack, None) \neq h (\text{Suc } z_0)$ 
    by simp
  have  $\forall t' \geq t. \text{fst } (\text{iterate } t' \text{ step } (?stack, None)) = []$ 
    using t iterate-step-empty-stack iterate-additive'[of t]
    by (metis le-Suc-ex prod.exhaust-sel)
  then have  $\forall t' \geq t. \text{iterate } t' \text{ step } (?stack, None) \neq h (\text{Suc } z_0)$ 
    using z-neq-Nil by auto
  then have  $\forall t'. \text{iterate } t' \text{ step } (?stack, None) \neq h (\text{Suc } z_0)$ 
    using not-h nat-le-linear by auto
  then have  $\neg \text{reachable } (?stack, None) (h (\text{Suc } z_0))$ 
    using reachable-def by simp
  then show False
    using reach-z[of Suc z0] h-def by simp
qed
then show ?thesis using diverg by simp
next
case diverg-f
let ?P =  $\lambda z. \text{eval } f (z \# xs) \uparrow$ 
define zmin where zmin  $\equiv \text{Least } ?P$ 
then have  $\forall y < zmin. \text{eval } f (y \# xs) \notin \{\text{None}, \text{Some } 0\}$ 
  using diverg-f eval-Mn-diverg[OF valid(2)] less-trans not-less-Least[of - ?P]
  by blast
moreover have f-zmin:  $\text{eval } f (zmin \# xs) \uparrow$ 
  using diverg-f LeastI-ex[of ?P] zmin-def by simp
ultimately have
  reachable (?stack, None) ((f, zmin # xs, []) # (Mn n f, xs, [zmin]) # rest, None)
    (is reachable - (?stack1, None))
  using reachable-Mn[OF assms] by simp
moreover have nonterminating (?stack1, None)
  using f-zmin assms valid diverg-f valid-ConsI by auto
ultimately have nonterminating (?stack, None)
  using reachable-nonterminating by simp
then show ?thesis using diverg-f by simp
next
case converg
then obtain z where z:  $\text{eval } (Mn n f) xs \downarrow = z$  by auto
have f-z:  $\text{eval } f (z \# xs) \downarrow = 0$ 
  and f-less-z:  $\bigwedge y. y < z \implies \text{eval } f (y \# xs) \downarrow \neq 0$ 
  using eval-Mn-convergE(2,3)[OF valid(2)] z by simp-all
then have
  reachable (?stack, None) ((f, z # xs, []) # (Mn n f, xs, [z]) # rest, None)
  using reachable-Mn[OF assms] by simp
also have reachable ... ((Mn n f, xs, [z]) # rest, eval f (z # xs))
  using assms(2)[of z # xs] valid f-z valid-ConsI correct-convergE
  by auto

```

```

    also have reachable ... (rest, Some z)
      using f-z f-less-z step-reachable by auto
    finally have reachable (?stack, None) (rest, Some z) .
    then show ?thesis using z by simp
  qed
qed

theorem step-correct:
  assumes valid ((f, xs, []) # rest)
  shows correct ((f, xs, []) # rest, None)
  using assms
proof (induction f arbitrary: xs rest)
  case Z
  then show ?case using valid-ConsE[of Z] step-reachable by auto
next
  case S
  then show ?case using valid-ConsE[of S] step-reachable by auto
next
  case (Id m n)
  then show ?case using valid-ConsE[of Id m n] by auto
next
  case Cn
  then show ?case using step-Cn-correct by presburger
next
  case Pr
  then show ?case using step-Pr-correct by simp
next
  case Mn
  then show ?case using step-Mn-correct by presburger
qed

```

1.5.2 Encoding partial recursive functions

In this section we define an injective, but not surjective, mapping from *recfs* to natural numbers.

abbreviation $\text{triple-encode} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{triple-encode } x \ y \ z \equiv \text{prod-encode } (x, \text{prod-encode } (y, z))$

abbreviation $\text{quad-encode} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{quad-encode } w \ x \ y \ z \equiv \text{prod-encode } (w, \text{prod-encode } (x, \text{prod-encode } (y, z)))$

fun $\text{encode} :: \text{recf} \Rightarrow \text{nat}$ **where**
 $\text{encode } Z = 0$
 $|\ \text{encode } S = 1$
 $|\ \text{encode } (\text{Id } m \ n) = \text{triple-encode } 2 \ m \ n$
 $|\ \text{encode } (\text{Cn } n \ f \ gs) = \text{quad-encode } 3 \ n \ (\text{encode } f) \ (\text{list-encode } (\text{map } \text{encode } gs))$
 $|\ \text{encode } (\text{Pr } n \ f \ g) = \text{quad-encode } 4 \ n \ (\text{encode } f) \ (\text{encode } g)$
 $|\ \text{encode } (\text{Mn } n \ f) = \text{triple-encode } 5 \ n \ (\text{encode } f)$

lemma $\text{prod-encode-gr1}: a > 1 \implies \text{prod-encode } (a, x) > 1$
using $\text{le-prod-encode-1 less-le-trans}$ **by** blast

lemma $\text{encode-not-Z-or-S}: \text{encode } f = \text{prod-encode } (a, b) \implies a > 1 \implies f \neq Z \wedge f \neq S$
by ($\text{metis encode.simps(1) encode.simps(2) less-numeral-extra(4) not-one-less-zero prod-encode-gr1}$)

```

lemma encode-injective: encode f = encode g  $\implies$  f = g
proof (induction g arbitrary: f)
  case Z
  have  $\bigwedge a x. a > 1 \implies \text{prod-encode } (a, x) > 0$ 
    using prod-encode-gr1 by (meson less-one less-trans)
  then have f  $\neq$  Z  $\implies$  encode f  $>$  0
    by (cases f) auto
  then have encode f = 0  $\implies$  f = Z by fastforce
  then show ?case using Z by simp
next
  case S
  have  $\bigwedge a x. a > 1 \implies \text{prod-encode } (a, x) \neq \text{Suc } 0$ 
    using prod-encode-gr1 by (metis One-nat-def less-numeral-extra(4))
  then have encode f = 1  $\implies$  f = S
    by (cases f) auto
  then show ?case using S by simp
next
  case Id
  then obtain z where *: encode f = prod-encode (2, z) by simp
  show ?case
    using Id by (cases f) (simp-all add: * encode-not-Z-or-S prod-encode-eq)
next
  case Cn
  then obtain z where *: encode f = prod-encode (3, z) by simp
  show ?case
  proof (cases f)
    case Z
    then show ?thesis using * encode-not-Z-or-S by simp
  next
    case S
    then show ?thesis using * encode-not-Z-or-S by simp
  next
    case Id
    then show ?thesis using * by (simp add: prod-encode-eq)
  next
    case Cn
    then show ?thesis
      using * Cn.IH Cn.premis list-decode-encode
      by (smt encode.simps(4) fst-conv list.inj-map-strong prod-encode-eq snd-conv)
  next
    case Pr
    then show ?thesis using * by (simp add: prod-encode-eq)
  next
    case Mn
    then show ?thesis using * by (simp add: prod-encode-eq)
  qed
next
  case Pr
  then obtain z where *: encode f = prod-encode (4, z) by simp
  show ?case
    using Pr by (cases f) (simp-all add: * encode-not-Z-or-S prod-encode-eq)
next
  case Mn
  then obtain z where *: encode f = prod-encode (5, z) by simp
  show ?case

```

using Mn **by** (*cases f*) (*simp-all add: * encode-not-Z-or-S prod-encode-eq*)
qed

definition *encode-kind* :: $\text{nat} \Rightarrow \text{nat}$ **where**
encode-kind $e \equiv$ if $e = 0$ then 0 else if $e = 1$ then 1 else *pdec1* e

lemma *encode-kind-0*: *encode-kind* (*encode* Z) = 0
unfolding *encode-kind-def* **by** *simp*

lemma *encode-kind-1*: *encode-kind* (*encode* S) = 1
unfolding *encode-kind-def* **by** *simp*

lemma *encode-kind-2*: *encode-kind* (*encode* ($\text{Id } m \ n$)) = 2
unfolding *encode-kind-def*
by (*metis encode.simps(1-3) encode-injective fst-conv prod-encode-inverse*
recf.simps(16) recf.simps(8))

lemma *encode-kind-3*: *encode-kind* (*encode* ($\text{Cn } n \ f \ gs$)) = 3
unfolding *encode-kind-def*
by (*metis encode.simps(1,2,4) encode-injective fst-conv prod-encode-inverse*
recf.simps(10) recf.simps(18))

lemma *encode-kind-4*: *encode-kind* (*encode* ($\text{Pr } n \ f \ g$)) = 4
unfolding *encode-kind-def*
by (*metis encode.simps(1,2,5) encode-injective fst-conv prod-encode-inverse*
recf.simps(12) recf.simps(20))

lemma *encode-kind-5*: *encode-kind* (*encode* ($\text{Mn } n \ f$)) = 5
unfolding *encode-kind-def*
by (*metis encode.simps(1,2,6) encode-injective fst-conv prod-encode-inverse*
recf.simps(14) recf.simps(22))

lemmas *encode-kind-n* =
encode-kind-0 encode-kind-1 encode-kind-2 encode-kind-3 encode-kind-4 encode-kind-5

lemma *encode-kind-Cn*:
assumes *encode-kind* (*encode* f) = 3
shows $\exists n \ f' \ gs. f = \text{Cn } n \ f' \ gs$
using *assms encode-kind-n* **by** (*cases f*) *auto*

lemma *encode-kind-Pr*:
assumes *encode-kind* (*encode* f) = 4
shows $\exists n \ f' \ g. f = \text{Pr } n \ f' \ g$
using *assms encode-kind-n* **by** (*cases f*) *auto*

lemma *encode-kind-Mn*:
assumes *encode-kind* (*encode* f) = 5
shows $\exists n \ g. f = \text{Mn } n \ g$
using *assms encode-kind-n* **by** (*cases f*) *auto*

lemma *pdec2-encode-Id*: *pdec2* (*encode* ($\text{Id } m \ n$)) = *prod-encode* (m, n)
by *simp*

lemma *pdec2-encode-Pr*: *pdec2* (*encode* ($\text{Pr } n \ f \ g$)) = *triple-encode* n (*encode* f) (*encode* g)
by *simp*

1.5.3 The step function on encoded configurations

In this section we construct a function $estep :: nat \Rightarrow nat$ that is equivalent to the function $step :: configuration \Rightarrow configuration$ except that it applies to encoded configurations. We start by defining an encoding for configurations.

definition $encode-frame :: frame \Rightarrow nat$ **where**
 $encode-frame\ s \equiv$
 $triple-encode\ (encode\ (fst\ s))\ (list-encode\ (fst\ (snd\ s)))\ (list-encode\ (snd\ (snd\ s)))$

lemma $encode-frame$:
 $encode-frame\ (f,\ xs,\ ls) = triple-encode\ (encode\ f)\ (list-encode\ xs)\ (list-encode\ ls)$
unfolding $encode-frame-def$ **by** $simp$

abbreviation $encode-option :: nat\ option \Rightarrow nat$ **where**
 $encode-option\ x \equiv if\ x = None\ then\ 0\ else\ Suc\ (the\ x)$

definition $encode-config :: configuration \Rightarrow nat$ **where**
 $encode-config\ cfg \equiv$
 $prod-encode\ (list-encode\ (map\ encode-frame\ (fst\ cfg)),\ encode-option\ (snd\ cfg))$

lemma $encode-config$:
 $encode-config\ (ss,\ rv) = prod-encode\ (list-encode\ (map\ encode-frame\ ss),\ encode-option\ rv)$
unfolding $encode-config-def$ **by** $simp$

Various projections from encoded configurations:

definition $e2stack$ **where** $e2stack\ e \equiv pdec1\ e$
definition $e2rv$ **where** $e2rv\ e \equiv pdec2\ e$
definition $e2tail$ **where** $e2tail\ e \equiv e-tl\ (e2stack\ e)$
definition $e2frame$ **where** $e2frame\ e \equiv e-hd\ (e2stack\ e)$
definition $e2i$ **where** $e2i\ e \equiv pdec1\ (e2frame\ e)$
definition $e2xs$ **where** $e2xs\ e \equiv pdec12\ (e2frame\ e)$
definition $e2ls$ **where** $e2ls\ e \equiv pdec22\ (e2frame\ e)$
definition $e2lenas$ **where** $e2lenas\ e \equiv e-length\ (e2xs\ e)$
definition $e2lenls$ **where** $e2lenls\ e \equiv e-length\ (e2ls\ e)$

lemma $e2rv-rv$ [$simp$]:
 $e2rv\ (encode-config\ (ss,\ rv)) = (if\ rv\ \uparrow\ then\ 0\ else\ Suc\ (the\ rv))$
unfolding $e2rv-def$ **using** $encode-config$ **by** $simp$

lemma $e2stack-stack$ [$simp$]:
 $e2stack\ (encode-config\ (ss,\ rv)) = list-encode\ (map\ encode-frame\ ss)$
unfolding $e2stack-def$ **using** $encode-config$ **by** $simp$

lemma $e2tail-tail$ [$simp$]:
 $e2tail\ (encode-config\ (s\ \# ss,\ rv)) = list-encode\ (map\ encode-frame\ ss)$
unfolding $e2tail-def$ **using** $encode-config$ **by** $fastforce$

lemma $e2frame-frame$ [$simp$]:
 $e2frame\ (encode-config\ (s\ \# ss,\ rv)) = encode-frame\ s$
unfolding $e2frame-def$ **using** $encode-config$ **by** $fastforce$

lemma $e2i-f$ [$simp$]:
 $e2i\ (encode-config\ ((f,\ xs,\ ls)\ \# ss,\ rv)) = encode\ f$
unfolding $e2i-def$ **using** $encode-config\ e2frame-frame\ encode-frame$ **by** $force$

lemma *e2xs-xs* [*simp*]:
 $e2xs$ (encode-config ((*f*, *xs*, *ls*) # *ss*, *rv*)) = list-encode *xs*
using *e2xs-def e2frame-frame encode-frame* **by** *force*

lemma *e2ls-ls* [*simp*]:
 $e2ls$ (encode-config ((*f*, *xs*, *ls*) # *ss*, *rv*)) = list-encode *ls*
using *e2ls-def e2frame-frame encode-frame* **by** *force*

lemma *e2lenas-lenas* [*simp*]:
 $e2lenas$ (encode-config ((*f*, *xs*, *ls*) # *ss*, *rv*)) = length *xs*
using *e2lenas-def e2frame-frame encode-frame* **by** *simp*

lemma *e2lenls-lenls* [*simp*]:
 $e2lenls$ (encode-config ((*f*, *xs*, *ls*) # *ss*, *rv*)) = length *ls*
using *e2lenls-def e2frame-frame encode-frame* **by** *simp*

lemma *e2stack-0-iff-Nil*:
assumes $e = \text{encode-config } (ss, rv)$
shows $e2stack\ e = 0 \iff ss = []$
using *assms*
by (*metis list-encode.simps(1) e2stack-stack list-encode-0 map-is-Nil-conv*)

lemma *e2ls-0-iff-Nil* [*simp*]: $\text{list-decode } (e2ls\ e) = [] \iff e2ls\ e = 0$
by (*metis list-decode.simps(1) list-encode-decode*)

We now define *eterm* piecemeal by considering the more complicated cases *Cn*, *Pr*, and *Mn* separately.

definition *estep-Cn* $e \equiv$
 if $e2lenls\ e = e\text{-length } (pdec222\ (e2i\ e))$
 then if $e2rv\ e = 0$
 then $\text{prod-encode } (e\text{-cons } (\text{triple-encode } (pdec122\ (e2i\ e))\ (e2ls\ e)\ 0)\ (e2stack\ e),\ 0)$
 else $\text{prod-encode } (e2tail\ e,\ e2rv\ e)$
 else if $e2rv\ e = 0$
 then if $e2lenls\ e < e\text{-length } (pdec222\ (e2i\ e))$
 then prod-encode
 ($e\text{-cons}$
 ($\text{triple-encode } (e\text{-nth } (pdec222\ (e2i\ e))\ (e2lenls\ e))\ (e2xs\ e)\ 0)$
 ($e2stack\ e$),
 0)
 else $\text{prod-encode } (e2tail\ e,\ e2rv\ e)$
 else prod-encode
 ($e\text{-cons}$
 ($\text{triple-encode } (e2i\ e)\ (e2xs\ e)\ (e\text{-snoc } (e2ls\ e)\ (e2rv\ e - 1))$)
 ($e2tail\ e$),
 0)

lemma *estep-Cn*:
assumes $c = (((Cn\ n\ f\ gs,\ xs,\ ls)\ #\ fs),\ rv)$
shows *estep-Cn* (encode-config *c*) = encode-config (*step* *c*)
using *encode-frame* **by** (*simp add: assms estep-Cn-def, simp add: encode-config assms*)

definition *estep-Pr* $e \equiv$
 if $e2ls\ e = 0$
 then if $e2rv\ e = 0$
 then prod-encode
 ($e\text{-cons } (\text{triple-encode } (pdec122\ (e2i\ e))\ (e\text{-tl } (e2xs\ e))\ 0)\ (e2stack\ e),$

```

    0)
  else prod-encode
    (e-cons (triple-encode (e2i e) (e2xs e) (singleton-encode (e2rv e - 1))) (e2tail e),
    0)
  else if e2lenls e = Suc (e-hd (e2xs e))
    then prod-encode (e2tail e, Suc (e-hd (e2ls e)))
  else if e2rv e = 0
    then prod-encode
      (e-cons
        (triple-encode
          (pdec222 (e2i e))
          (e-cons (e2lenls e - 1) (e-cons (e-hd (e2ls e)) (e-tl (e2xs e))))
          0)
        (e2stack e),
        0)
    else prod-encode
      (e-cons
        (triple-encode (e2i e) (e2xs e) (e-cons (e2rv e - 1) (e2ls e))) (e2tail e),
        0)

```

lemma *estep-Pr1*:

```

assumes c = (((Pr n f g, xs, ls) # fs), rv)
and ls ≠ []
and length ls ≠ Suc (hd xs)
and rv ≠ None
and recfn (length xs) (Pr n f g)
shows estep-Pr (encode-config c) = encode-config (step c)

```

proof –

```

let ?e = encode-config c
from assms(5) have length xs > 0 by auto
then have eq: hd xs = e-hd (e2xs ?e)
  using assms e-hd-def by auto
have step c = ((Pr n f g, xs, (the rv) # ls) # fs, None)
  (is step c = (?t # ?ss, None))
  using assms by simp
then have encode-config (step c) =
  prod-encode (list-encode (map encode-frame (?t # ?ss)), 0)
  using encode-config by simp
also have ... =
  prod-encode (e-cons (encode-frame ?t) (list-encode (map encode-frame (?ss))), 0)
  by simp
also have ... = prod-encode (e-cons (encode-frame ?t) (e2tail ?e), 0)
  using assms(1) by simp
also have ... = prod-encode
  (e-cons
    (triple-encode (e2i ?e) (e2xs ?e) (e-cons (e2rv ?e - 1) (e2ls ?e)))
    (e2tail ?e),
    0)
  by (simp add: assms encode-frame)
finally show ?thesis
  using assms eq estep-Pr-def by auto
qed

```

lemma *estep-Pr2*:

```

assumes c = (((Pr n f g, xs, ls) # fs), rv)
and ls ≠ []

```

```

    and length ls ≠ Suc (hd xs)
    and rv = None
    and recfn (length xs) (Pr n f g)
  shows estep-Pr (encode-config c) = encode-config (step c)
proof -
  let ?e = encode-config c
  from assms(5) have length xs > 0 by auto
  then have eq: hd xs = e-hd (e2xs ?e)
    using assms e-hd-def by auto
  have step c = ((g, (length ls - 1) # hd ls # tl xs, []) # (Pr n f g, xs, ls) # fs, None)
    (is step c = (?t # ?ss, None))
    using assms by simp
  then have encode-config (step c) =
    prod-encode (list-encode (map encode-frame (?t # ?ss)), 0)
    using encode-config by simp
  also have ... =
    prod-encode (e-cons (encode-frame ?t) (list-encode (map encode-frame (?ss))), 0)
    by simp
  also have ... = prod-encode (e-cons (encode-frame ?t) (e2stack ?e), 0)
    using assms(1) by simp
  also have ... = prod-encode
    (e-cons
      (triple-encode
        (pdec222 (e2i ?e))
        (e-cons (e2lenls ?e - 1) (e-cons (e-hd (e2ls ?e)) (e-tl (e2xs ?e))))
        0)
      (e2stack ?e),
      0)
    using assms(1,2) encode-frame[of g (length ls - 1) # hd ls # tl xs []]
      pdec2-encode-Pr[of n f g] e2xs-xs e2i-f e2lenls-lenls e2ls-ls e-hd
    by (metis list-encode.simps(1) list.collapse list-decode-encode
      prod-encode-inverse snd-conv)
  finally show ?thesis
    using assms eq estep-Pr-def by auto
qed

```

lemma estep-Pr3:

```

  assumes c = (((Pr n f g, xs, ls) # fs), rv)
  and ls ≠ []
  and length ls = Suc (hd xs)
  and recfn (length xs) (Pr n f g)
  shows estep-Pr (encode-config c) = encode-config (step c)
proof -
  let ?e = encode-config c
  from assms(4) have length xs > 0 by auto
  then have hd xs = e-hd (e2xs ?e)
    using assms e-hd-def by auto
  then have (length ls = Suc (hd xs)) = (e2lenls ?e = Suc (e-hd (e2xs ?e)))
    using assms by simp
  then have *: estep-Pr ?e = prod-encode (e2tail ?e, Suc (e-hd (e2ls ?e)))
    using assms estep-Pr-def by auto
  have step c = (fs, Some (hd ls))
    using assms(1,2,3) by simp
  then have encode-config (step c) =
    prod-encode (list-encode (map encode-frame fs), encode-option (Some (hd ls)))
    using encode-config by simp

```

also have ... =
 $\text{prod-encode } (\text{list-encode } (\text{map } \text{encode-frame } fs), \text{ encode-option } (\text{Some } (\text{e-hd } (e2ls ?e))))$
using *assms(1,2) e-hd-def* **by** *auto*
also have ... = $\text{prod-encode } (\text{list-encode } (\text{map } \text{encode-frame } fs), \text{ Suc } (\text{e-hd } (e2ls ?e)))$
by *simp*
also have ... = $\text{prod-encode } (e2tail ?e, \text{ Suc } (\text{e-hd } (e2ls ?e)))$
using *assms(1)* **by** *simp*
finally have $\text{encode-config } (\text{step } c) = \text{prod-encode } (e2tail ?e, \text{ Suc } (\text{e-hd } (e2ls ?e)))$.
then show *?thesis*
using *estep-Pr-def ** **by** *presburger*
qed

lemma *estep-Pr4*:
assumes $c = (((Pr\ n\ f\ g, xs, ls) \# fs), rv)$ **and** $ls = []$
shows $\text{estep-Pr } (\text{encode-config } c) = \text{encode-config } (\text{step } c)$
using *encode-frame*
by (*simp add: assms estep-Pr-def, simp add: encode-config assms*)

lemma *estep-Pr*:
assumes $c = (((Pr\ n\ f\ g, xs, ls) \# fs), rv)$
and *recfn* (*length* *xs*) (*Pr n f g*)
shows $\text{estep-Pr } (\text{encode-config } c) = \text{encode-config } (\text{step } c)$
using *assms estep-Pr1 estep-Pr2 estep-Pr3 estep-Pr4* **by** *auto*

definition *estep-Mn e* \equiv
if $e2ls\ e = 0$
then *prod-encode*
 $(e\text{-cons } (\text{triple-encode } (\text{pdec22 } (e2i\ e)) (e\text{-cons } 0 (e2xs\ e))\ 0) (e\text{-cons } (\text{triple-encode } (e2i\ e) (e2xs\ e) (\text{singleton-encode } 0)) (e2tail\ e)), 0)$
else if $e2rv\ e = 1$
then *prod-encode* $(e2tail\ e, \text{Suc } (\text{e-hd } (e2ls\ e)))$
else *prod-encode*
 $(e\text{-cons } (\text{triple-encode } (\text{pdec22 } (e2i\ e)) (e\text{-cons } (\text{Suc } (\text{e-hd } (e2ls\ e))) (e2xs\ e))\ 0) (e\text{-cons } (\text{triple-encode } (e2i\ e) (e2xs\ e) (\text{singleton-encode } (\text{Suc } (\text{e-hd } (e2ls\ e))))) (e2tail\ e)), 0)$

lemma *estep-Mn*:
assumes $c = (((Mn\ n\ f, xs, ls) \# fs), rv)$
shows $\text{estep-Mn } (\text{encode-config } c) = \text{encode-config } (\text{step } c)$
proof –
let $?e = \text{encode-config } c$
consider $ls \neq []$ **and** $rv \neq \text{Some } 0 \mid ls \neq []$ **and** $rv = \text{Some } 0 \mid ls = []$
by *auto*
then show *?thesis*
proof (*cases*)
case *1*
then have *step-c: step c =*
 $(f, (\text{Suc } (\text{hd } ls)) \# xs, []) \# (Mn\ n\ f, xs, [\text{Suc } (\text{hd } ls)]) \# fs, \text{None}$
(is $\text{step } c = ?cfc$ **)**

```

using assms by simp
have estep-Mn ?e =
  prod-encode
    (e-cons
      (triple-encode (encode f) (e-cons (Suc (hd ls)) (list-encode xs)) 0)
      (e-cons
        (triple-encode (encode (Mn n f)) (list-encode xs) (singleton-encode (Suc (hd ls))))
        (list-encode (map encode-frame fs))),
    0)
using 1 assms e-hd-def estep-Mn-def by auto
also have ... = encode-config ?cfg
using encode-config by (simp add: encode-frame)
finally show ?thesis
using step-c by simp
next
case 2
have estep-Mn ?e = prod-encode (e2tail ?e, Suc (e-hd (e2ls ?e)))
using 2 assms estep-Mn-def by auto
also have ... = prod-encode (e2tail ?e, Suc (hd ls))
using 2 assms e-hd-def by auto
also have ... = prod-encode (list-encode (map encode-frame fs), Suc (hd ls))
using assms by simp
also have ... = encode-config (fs, Some (hd ls))
using encode-config by simp
finally show ?thesis
using 2 assms by simp
next
case 3
then show ?thesis
using assms encode-frame by (simp add: estep-Mn-def, simp add: encode-config)
qed
qed

```

definition *estep e* \equiv

```

if e2stack e = 0 then prod-encode (0, e2rv e)
else if e2i e = 0 then prod-encode (e2tail e, 1)
else if e2i e = 1 then prod-encode (e2tail e, Suc (Suc (e-hd (e2xs e))))
else if encode-kind (e2i e) = 2 then
  prod-encode (e2tail e, Suc (e-nth (e2xs e) (pdec22 (e2i e))))
else if encode-kind (e2i e) = 3 then estep-Cn e
else if encode-kind (e2i e) = 4 then estep-Pr e
else if encode-kind (e2i e) = 5 then estep-Mn e
else 0

```

lemma *estep-Z*:

```

assumes c = (((Z, xs, ls) # fs), rv)
shows estep (encode-config c) = encode-config (step c)
using encode-frame by (simp add: assms estep-def, simp add: encode-config assms)

```

lemma *estep-S*:

```

assumes c = (((S, xs, ls) # fs), rv)
and recfn (length xs) (fst (hd (fst c)))
shows estep (encode-config c) = encode-config (step c)

```

proof –

```

let ?e = encode-config c
from assms have length xs > 0 by auto

```

```

then have eq: hd xs = e-hd (e2xs ?e)
  using assms(1) e-hd-def by auto
then have estep ?e = prod-encode (e2tail ?e, Suc (Suc (e-hd (e2xs ?e))))
  using assms(1) estep-def by simp
moreover have step c = (fs, Some (Suc (hd xs)))
  using assms(1) by simp
ultimately show ?thesis
  using assms(1) eq estep-def encode-config[of fs Some (Suc (hd xs))] by simp
qed

```

lemma *estep-Id*:

```

assumes c = (((Id m n, xs, ls) # fs), rv)
  and recfn (length xs) (fst (hd (fst c)))
shows estep (encode-config c) = encode-config (step c)
proof -
let ?e = encode-config c
from assms have length xs = m and m > 0 by auto
then have eq: xs ! n = e-nth (e2xs ?e) n
  using assms e-hd-def by auto
moreover have encode-kind (e2i ?e) = 2
  using assms(1) encode-kind-2 by auto
ultimately have estep ?e =
  prod-encode (e2tail ?e, Suc (e-nth (e2xs ?e) (pdec22 (e2i ?e))))
  using assms estep-def encode-kind-def by auto
moreover have step c = (fs, Some (xs ! n))
  using assms(1) by simp
ultimately show ?thesis
  using assms(1) eq encode-config[of fs Some (xs ! n)] by simp
qed

```

lemma *estep*:

```

assumes valid (fst c)
shows estep (encode-config c) = encode-config (step c)
proof (cases fst c)
case Nil
then show ?thesis
  using estep-def
  by (metis list-encode.simps(1) e2rv-def e2stack-stack encode-config-def
    map-is-Nil-conv prod.collapse prod-encode-inverse snd-conv step.simps(1))
next
case (Cons s fs)
then obtain f xs ls rv where c: c = ((f, xs, ls) # fs, rv)
  by (metis prod.exhaust-sel)
with assms valid-def have lenas: recfn (length xs) f by simp
show ?thesis
proof (cases f)
case Z
then show ?thesis using estep-Z c by simp
next
case S
then show ?thesis using estep-S c lenas by simp
next
case Id
then show ?thesis using estep-Id c lenas by simp
next
case Cn

```

```

then show ?thesis
  using estep-Cn c
  by (metis e2i-f e2stack-0-iff-Nil encode.simps(1) encode.simps(2) encode-kind-2
    encode-kind-3 encode-kind-Cn estep-def list.distinct(1) recf.distinct(13)
    recf.distinct(19) recf.distinct(5))
next
  case Pr
  then show ?thesis
    using estep-Pr c lenas
    by (metis e2i-f e2stack-0-iff-Nil encode.simps(1) encode.simps(2) encode-kind-2
      encode-kind-4 encode-kind-Cn encode-kind-Pr estep-def list.distinct(1) recf.distinct(15)
      recf.distinct(21) recf.distinct(25) recf.distinct(7))
next
  case Mn
  then show ?thesis
    using estep-Pr c lenas
    by (metis (no-types, lifting) e2i-f e2stack-0-iff-Nil encode.simps(1)
      encode.simps(2) encode-kind-2 encode-kind-5 encode-kind-Cn encode-kind-Mn encode-kind-Pr
      estep-Mn estep-def list.distinct(1) recf.distinct(17) recf.distinct(23)
      recf.distinct(27) recf.distinct(9))
qed
qed

```

1.5.4 The step function as a partial recursive function

In this section we construct a primitive recursive function $r\text{-step}$ computing $estep$. This will entail defining $recfs$ for many functions defined in the previous section.

definition $r\text{-e2stack} \equiv r\text{-pdec1}$

lemma $r\text{-e2stack-prim}$: $\text{prim-recfn } 1 \text{ } r\text{-e2stack}$
unfolding $r\text{-e2stack-def}$ **using** $r\text{-pdec1-prim}$ **by** simp

lemma $r\text{-e2stack}$ [simp]: $\text{eval } r\text{-e2stack } [e] \downarrow = e2stack \ e$
unfolding $r\text{-e2stack-def}$ $e2stack\text{-def}$ **using** $r\text{-pdec1-prim}$ **by** simp

definition $r\text{-e2rv} \equiv r\text{-pdec2}$

lemma $r\text{-e2rv-prim}$: $\text{prim-recfn } 1 \text{ } r\text{-e2rv}$
unfolding $r\text{-e2rv-def}$ **using** $r\text{-pdec2-prim}$ **by** simp

lemma $r\text{-e2rv}$ [simp]: $\text{eval } r\text{-e2rv } [e] \downarrow = e2rv \ e$
unfolding $r\text{-e2rv-def}$ $e2rv\text{-def}$ **using** $r\text{-pdec2-prim}$ **by** simp

definition $r\text{-e2tail} \equiv Cn \ 1 \ r\text{-tl } [r\text{-e2stack}]$

lemma $r\text{-e2tail-prim}$: $\text{prim-recfn } 1 \text{ } r\text{-e2tail}$
unfolding $r\text{-e2tail-def}$ **using** $r\text{-e2stack-prim}$ $r\text{-tl-prim}$ **by** simp

lemma $r\text{-e2tail}$ [simp]: $\text{eval } r\text{-e2tail } [e] \downarrow = e2tail \ e$
unfolding $r\text{-e2tail-def}$ $e2tail\text{-def}$ **using** $r\text{-e2stack-prim}$ $r\text{-tl-prim}$ **by** simp

definition $r\text{-e2frame} \equiv Cn \ 1 \ r\text{-hd } [r\text{-e2stack}]$

lemma $r\text{-e2frame-prim}$: $\text{prim-recfn } 1 \text{ } r\text{-e2frame}$
unfolding $r\text{-e2frame-def}$ **using** $r\text{-hd-prim}$ $r\text{-e2stack-prim}$ **by** simp

lemma *r-e2frame* [*simp*]: *eval r-e2frame* [e] \Downarrow *e2frame* e
unfolding *r-e2frame-def* *e2frame-def* **using** *r-hd-prim* *r-e2stack-prim* **by** *simp*

definition *r-e2i* \equiv *Cn 1 r-pdec1* [*r-e2frame*]

lemma *r-e2i-prim*: *prim-recfn 1 r-e2i*
unfolding *r-e2i-def* **using** *r-pdec12-prim* *r-e2frame-prim* **by** *simp*

lemma *r-e2i* [*simp*]: *eval r-e2i* [e] \Downarrow *e2i* e
unfolding *r-e2i-def* *e2i-def* **using** *r-pdec12-prim* *r-e2frame-prim* **by** *simp*

definition *r-e2xs* \equiv *Cn 1 r-pdec12* [*r-e2frame*]

lemma *r-e2xs-prim*: *prim-recfn 1 r-e2xs*
unfolding *r-e2xs-def* **using** *r-pdec122-prim* *r-e2frame-prim* **by** *simp*

lemma *r-e2xs* [*simp*]: *eval r-e2xs* [e] \Downarrow *e2xs* e
unfolding *r-e2xs-def* *e2xs-def* **using** *r-pdec122-prim* *r-e2frame-prim* **by** *simp*

definition *r-e2ls* \equiv *Cn 1 r-pdec22* [*r-e2frame*]

lemma *r-e2ls-prim*: *prim-recfn 1 r-e2ls*
unfolding *r-e2ls-def* **using** *r-pdec222-prim* *r-e2frame-prim* **by** *simp*

lemma *r-e2ls* [*simp*]: *eval r-e2ls* [e] \Downarrow *e2ls* e
unfolding *r-e2ls-def* *e2ls-def* **using** *r-pdec222-prim* *r-e2frame-prim* **by** *simp*

definition *r-e2lenls* \equiv *Cn 1 r-length* [*r-e2ls*]

lemma *r-e2lenls-prim*: *prim-recfn 1 r-e2lenls*
unfolding *r-e2lenls-def* **using** *r-length-prim* *r-e2ls-prim* **by** *simp*

lemma *r-e2lenls* [*simp*]: *eval r-e2lenls* [e] \Downarrow *e2lenls* e
unfolding *r-e2lenls-def* *e2lenls-def* **using** *r-length-prim* *r-e2ls-prim* **by** *simp*

definition *r-kind* \equiv
Cn 1 r-ifz [*Id 1 0*, *Z*, *Cn 1 r-ifeq* [*Id 1 0*, *r-const 1*, *r-const 1*, *r-pdec1*]]

lemma *r-kind-prim*: *prim-recfn 1 r-kind*
unfolding *r-kind-def* **by** *simp*

lemma *r-kind*: *eval r-kind* [e] \Downarrow *encode-kind* e
unfolding *r-kind-def* *encode-kind-def* **by** *simp*

lemmas *helpers-for-r-step-prim* =
r-e2i-prim
r-e2lenls-prim
r-e2ls-prim
r-e2rv-prim
r-e2xs-prim
r-e2stack-prim
r-e2tail-prim
r-e2frame-prim

We define primitive recursive functions *r-step-Id*, *r-step-Cn*, *r-step-Pr*, and *r-step-Mn*.

The last three correspond to *estep-Cn*, *estep-Pr*, and *estep-Mn* from the previous section.

definition *r-step-Id* \equiv

Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 r-nth [r-e2xs, Cn 1 r-pdec22 [r-e2i]]]]

lemma *r-step-Id*:

eval r-step-Id [e] \downarrow = prod-encode (e2tail e, Suc (e-nth (e2xs e) (pdec22 (e2i e))))
unfolding *r-step-Id-def* **using** *helpers-for-r-step-prim* **by** *simp*

abbreviation *r-triple-encode* $::$ *recf \Rightarrow recf \Rightarrow recf \Rightarrow recf* **where**

r-triple-encode x y z \equiv Cn 1 r-prod-encode [x, Cn 1 r-prod-encode [y, z]]

definition *r-step-Cn* \equiv

Cn 1 r-ifeq
[r-e2lenls,
Cn 1 r-length [Cn 1 r-pdec222 [r-e2i]],
Cn 1 r-ifz
[r-e2rv,
Cn 1 r-prod-encode
[Cn 1 r-cons [r-triple-encode (Cn 1 r-pdec122 [r-e2i]) r-e2ls Z, r-e2stack],
Z],
Cn 1 r-prod-encode [r-e2tail, r-e2rv]],
Cn 1 r-ifz
[r-e2rv,
Cn 1 r-iftless
[r-e2lenls,
Cn 1 r-length [Cn 1 r-pdec222 [r-e2i]],
Cn 1 r-prod-encode
[Cn 1 r-cons
[r-triple-encode (Cn 1 r-nth [Cn 1 r-pdec222 [r-e2i], r-e2lenls]) r-e2xs Z,
r-e2stack],
Z],
Cn 1 r-prod-encode [r-e2tail, r-e2rv]],
Cn 1 r-prod-encode
[Cn 1 r-cons
[r-triple-encode r-e2i r-e2xs (Cn 1 r-snoc [r-e2ls, Cn 1 r-dec [r-e2rv]],
r-e2tail],
Z]]])

lemma *r-step-Cn-prim*: *prim-recfn 1 r-step-Cn*

unfolding *r-step-Cn-def* **using** *helpers-for-r-step-prim* **by** *simp*

lemma *r-step-Cn*: *eval r-step-Cn [e] \downarrow = estep-Cn e*

unfolding *r-step-Cn-def* *estep-Cn-def* **using** *helpers-for-r-step-prim* **by** *simp*

definition *r-step-Pr* \equiv

Cn 1 r-ifz
[r-e2ls,
Cn 1 r-ifz
[r-e2rv,
Cn 1 r-prod-encode
[Cn 1 r-cons
[r-triple-encode (Cn 1 r-pdec122 [r-e2i]) (Cn 1 r-tl [r-e2xs]) Z,
r-e2stack],
Z],
Cn 1 r-prod-encode

```

[Cn 1 r-cons
 [r-triple-encode r-e2i r-e2xs (Cn 1 r-singleton-encode [Cn 1 r-dec [r-e2rv]]),
  r-e2tail],
 Z]],
Cn 1 r-ifeq
[r-e2lens,
 Cn 1 S [Cn 1 r-hd [r-e2xs]],
 Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 r-hd [r-e2ls]]],
 Cn 1 r-ifz
 [r-e2rv,
  Cn 1 r-prod-encode
  [Cn 1 r-cons
   [r-triple-encode
    (Cn 1 r-pdec222 [r-e2i])
    (Cn 1 r-cons
     [Cn 1 r-dec [r-e2lens],
      Cn 1 r-cons [Cn 1 r-hd [r-e2ls],
       Cn 1 r-tl [r-e2xs]])]
   Z,
   r-e2stack],
  Z],
 Cn 1 r-prod-encode
 [Cn 1 r-cons
  [r-triple-encode r-e2i r-e2xs (Cn 1 r-cons [Cn 1 r-dec [r-e2rv], r-e2ls]),
   r-e2tail],
  Z]]]]

```

lemma *r-step-Pr-prim: prim-recfn 1 r-step-Pr*
unfolding *r-step-Pr-def* **using** *helpers-for-r-step-prim* **by** *simp*

lemma *r-step-Pr: eval r-step-Pr [e] ↓ = estep-Pr e*
unfolding *r-step-Pr-def estep-Pr-def* **using** *helpers-for-r-step-prim* **by** *simp*

definition *r-step-Mn* ≡

```

Cn 1 r-ifz
[r-e2ls,
 Cn 1 r-prod-encode
 [Cn 1 r-cons
  [r-triple-encode (Cn 1 r-pdec22 [r-e2i]) (Cn 1 r-cons [Z, r-e2xs]) Z,
   Cn 1 r-cons
   [r-triple-encode r-e2i r-e2xs (Cn 1 r-singleton-encode [Z]),
    r-e2tail]],
 Z],
Cn 1 r-ifeq
[r-e2rv,
 r-const 1,
 Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 r-hd [r-e2ls]]],
 Cn 1 r-prod-encode
 [Cn 1 r-cons
  [r-triple-encode
   (Cn 1 r-pdec22 [r-e2i])
   (Cn 1 r-cons [Cn 1 S [Cn 1 r-hd [r-e2ls]], r-e2xs])
  Z,
  Cn 1 r-cons
  [r-triple-encode r-e2i r-e2xs (Cn 1 r-singleton-encode [Cn 1 S [Cn 1 r-hd [r-e2ls]]]),
   r-e2tail]],

```

$Z]]]$

lemma *r-step-Mn-prim: prim-recfn 1 r-step-Mn*
unfolding *r-step-Mn-def using helpers-for-r-step-prim by simp*

lemma *r-step-Mn: eval r-step-Mn [e] \downarrow = estep-Mn e*
unfolding *r-step-Mn-def estep-Mn-def using helpers-for-r-step-prim by simp*

definition *r-step \equiv*
 Cn 1 r-ifz
 [*r-e2stack,*
 Cn 1 r-prod-encode [Z, r-e2rv],
 Cn 1 r-ifz
 [*r-e2i,*
 Cn 1 r-prod-encode [r-e2tail, r-const 1],
 Cn 1 r-ifeq
 [*r-e2i,*
 r-const 1,
 Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 S [Cn 1 r-hd [r-e2xs]]]]],
 Cn 1 r-ifeq
 [*Cn 1 r-kind [r-e2i],*
 r-const 2,
 Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 r-nth [r-e2xs, Cn 1 r-pdec22 [r-e2i]]]]],
 Cn 1 r-ifeq
 [*Cn 1 r-kind [r-e2i],*
 r-const 3,
 r-step-Cn,
 Cn 1 r-ifeq
 [*Cn 1 r-kind [r-e2i],*
 r-const 4,
 r-step-Pr,
 Cn 1 r-ifeq
 [*Cn 1 r-kind [r-e2i], r-const 5, r-step-Mn, Z]]]]]]]]]*

lemma *r-step-prim: prim-recfn 1 r-step*
unfolding *r-step-def*
using *r-kind-prim r-step-Mn-prim r-step-Pr-prim r-step-Cn-prim helpers-for-r-step-prim*
by *simp*

lemma *r-step: eval r-step [e] \downarrow = estep e*
unfolding *r-step-def estep-def*
using *r-kind-prim r-step-Mn-prim r-step-Pr-prim r-step-Cn-prim helpers-for-r-step-prim*
 r-kind r-step-Cn r-step-Pr r-step-Mn
by *simp*

theorem *r-step-equiv-step:*
 assumes *valid (fst c)*
 shows *eval r-step [encode-config c] \downarrow = encode-config (step c)*
 using *r-step estep assms by simp*

1.5.5 The universal function

The next function computes the configuration after arbitrarily many steps.

definition *r-leap \equiv*
 Pr 2

(Cn 2 r-prod-encode
 [Cn 2 r-singleton-encode
 [Cn 2 r-prod-encode [Id 2 0], Cn 2 r-prod-encode [Id 2 1, r-constn 1 0]],
 r-constn 1 0])
 (Cn 4 r-step [Id 4 1])

lemma *r-leap-prim* [simp]: prim-recfn 3 *r-leap*
 unfolding *r-leap-def* using *r-step-prim* by *simp*

lemma *r-leap-total*: eval *r-leap* [t, i, x] ↓
 using prim-recfn-total[OF *r-leap-prim*] by *simp*

lemma *r-leap*:
 assumes *i = encode f and recfn (e-length x) f*
 shows eval *r-leap* [t, i, x] ↓= encode-config (iterate t step (([f, list-decode x, []]), None))
proof (induction t)
 case 0
 then show ?case
 unfolding *r-leap-def* using *r-step-prim* assms encode-config encode-frame by *simp*
next
 case (Suc t)
 let ?c = ([f, list-decode x, []]), None)
 let ?tc = iterate t step ?c
 have valid (fst ?c)
 using valid-def assms by *simp*
 then have valid: valid (fst ?tc)
 using iterate-step-valid by *simp*
 have eval *r-leap* [Suc t, i, x] =
 eval (Cn 4 r-step [Id 4 1]) [t, the (eval *r-leap* [t, i, x]), i, x]
 by (smt One-nat-def Suc-eq-plus1 eq-numeral-Suc eval-Pr-converg-Suc list.size(3) list.size(4)
 nat-1-add-1 pred-numeral-simps(3) *r-leap-def r-leap-prim r-leap-total*)
 then have eval *r-leap* [Suc t, i, x] = eval (Cn 4 r-step [Id 4 1]) [t, encode-config ?tc, i, x]
 using *Suc* by *simp*
 then have eval *r-leap* [Suc t, i, x] = eval *r-step* [encode-config ?tc]
 using *r-step-prim* by *simp*
 then have eval *r-leap* [Suc t, i, x] ↓= encode-config (step ?tc)
 by (simp add: *r-step-equiv-step valid*)
 then show ?case by *simp*
qed

lemma *step-leaves-empty-stack-empty*:
 assumes iterate t step (([f, list-decode x, []]), None) = ([], Some v)
 shows iterate (t + t') step (([f, list-decode x, []]), None) = ([], Some v)
 using assms by (induction t') *simp-all*

The next function is essentially a convenience wrapper around *r-leap*. It returns zero if the configuration returned by *r-leap* is non-final, and *Suc v* if the configuration is final with return value *v*.

definition *r-result* ≡
 Cn 3 r-ifz [Cn 3 r-pdec1 [r-leap], Cn 3 r-pdec2 [r-leap], r-constn 2 0]

lemma *r-result-prim* [simp]: prim-recfn 3 *r-result*
 unfolding *r-result-def* using *r-leap-prim* by *simp*

lemma *r-result-total*: total *r-result*
 using *r-result-prim* by *blast*

lemma *r-result-empty-stack-None*:

assumes $i = \text{encode } f$
and $\text{recfn } (e\text{-length } x) f$
and $\text{iterate } t \text{ step } ((f, \text{list-decode } x, []), \text{None}) = ([], \text{None})$
shows $\text{eval } r\text{-result } [t, i, x] \Downarrow = 0$
unfolding *r-result-def*
using *assms r-leap e2stack-0-iff-Nil e2stack-def e2stack-stack r-leap-total r-leap-prim e2rv-def e2rv-rv*
by *simp*

lemma *r-result-empty-stack-Some*:

assumes $i = \text{encode } f$
and $\text{recfn } (e\text{-length } x) f$
and $\text{iterate } t \text{ step } ((f, \text{list-decode } x, []), \text{None}) = ([], \text{Some } v)$
shows $\text{eval } r\text{-result } [t, i, x] \Downarrow = \text{Suc } v$
unfolding *r-result-def*
using *assms r-leap e2stack-0-iff-Nil e2stack-def e2stack-stack r-leap-total r-leap-prim e2rv-def e2rv-rv*
by *simp*

lemma *r-result-empty-stack-stays*:

assumes $i = \text{encode } f$
and $\text{recfn } (e\text{-length } x) f$
and $\text{iterate } t \text{ step } ((f, \text{list-decode } x, []), \text{None}) = ([], \text{Some } v)$
shows $\text{eval } r\text{-result } [t + t', i, x] \Downarrow = \text{Suc } v$
using *assms step-leaves-empty-stack-empty r-result-empty-stack-Some* **by** *simp*

lemma *r-result-nonempty-stack*:

assumes $i = \text{encode } f$
and $\text{recfn } (e\text{-length } x) f$
and $\text{fst } (\text{iterate } t \text{ step } ((f, \text{list-decode } x, []), \text{None})) \neq []$
shows $\text{eval } r\text{-result } [t, i, x] \Downarrow = 0$

proof –

obtain $ss \ rv$ **where** $\text{iterate } t \text{ step } ((f, \text{list-decode } x, []), \text{None}) = (ss, rv)$
by *fastforce*
moreover from *this* **assms**(3) **have** $ss \neq []$ **by** *simp*
ultimately have $\text{eval } r\text{-leap } [t, i, x] \Downarrow = \text{encode-config } (ss, rv)$
using *assms r-leap* **by** *simp*
then have $\text{eval } (Cn \ 3 \ r\text{-pdec1 } [r\text{-leap}]) [t, i, x] \Downarrow \neq 0$
using $\langle ss \neq [] \rangle$ *r-leap-prim encode-config r-leap-total list-encode-0* **by** *auto*
then show *?thesis* **unfolding** *r-result-def* **using** *r-leap-prim* **by** *auto*
qed

lemma *r-result-Suc*:

assumes $i = \text{encode } f$
and $\text{recfn } (e\text{-length } x) f$
and $\text{eval } r\text{-result } [t, i, x] \Downarrow = \text{Suc } v$
shows $\text{iterate } t \text{ step } ((f, \text{list-decode } x, []), \text{None}) = ([], \text{Some } v)$
(is *?cfg = -***)**

proof (*cases fst ?cfg*)

case *Nil*

then show *?thesis*

using *assms r-result-empty-stack-None r-result-empty-stack-Some*

by (*metis Zero-not-Suc nat.inject option.collapse option.inject prod.exhaust-sel*)

next

case *Cons*
then show *?thesis* **using** *assms r-result-nonempty-stack* **by** *simp*
qed

lemma *r-result-converg*:

assumes $i = \text{encode } f$
and $\text{recfn } (e\text{-length } x) f$
and $\text{eval } f \text{ (list-decode } x) \Downarrow = v$
shows $\exists t.$
 $(\forall t' \geq t. \text{eval } r\text{-result } [t', i, x] \Downarrow = \text{Suc } v) \wedge$
 $(\forall t' < t. \text{eval } r\text{-result } [t', i, x] \Downarrow = 0)$

proof –

let $?xs = \text{list-decode } x$
let $?stack = [(f, ?xs, [])]$
have $\text{wellf } f$ **using** *assms(2)* **by** *simp*
moreover have $\text{length } ?xs = \text{arity } f$
using *assms(2)* **by** *simp*
ultimately have $\text{correct } (?stack, \text{None})$
using *step-correct valid-def* **by** *simp*
with *assms(3)* **have** $\text{reachable } (?stack, \text{None}) ([], \text{Some } v)$
by *simp*
then obtain t **where**
 $\text{iterate } t \text{ step } (?stack, \text{None}) = ([], \text{Some } v)$
 $\forall t' < t. \text{fst } (\text{iterate } t' \text{ step } (?stack, \text{None})) \neq []$
using *reachable-iterate-step-empty-stack* **by** *blast*
then have t :
 $\text{eval } r\text{-result } [t, i, x] \Downarrow = \text{Suc } v$
 $\forall t' < t. \text{eval } r\text{-result } [t', i, x] \Downarrow = 0$
using *r-result-empty-stack-Some r-result-nonempty-stack assms(1,2)*
by *simp-all*
then have $\text{eval } r\text{-result } [t + t', i, x] \Downarrow = \text{Suc } v$ **for** t'
using *r-result-empty-stack-stays assms r-result-Suc* **by** *simp*
then have $\forall t' \geq t. \text{eval } r\text{-result } [t', i, x] \Downarrow = \text{Suc } v$
using *le-Suc-ex* **by** *blast*
with $t(2)$ **show** *?thesis* **by** *auto*
qed

lemma *r-result-diverg*:

assumes $i = \text{encode } f$
and $\text{recfn } (e\text{-length } x) f$
and $\text{eval } f \text{ (list-decode } x) \Uparrow$
shows $\text{eval } r\text{-result } [t, i, x] \Downarrow = 0$

proof –

let $?xs = \text{list-decode } x$
let $?stack = [(f, ?xs, [])]$
have $\text{recfn } (\text{length } ?xs) f$
using *assms(2)* **by** *auto*
then have $\text{correct } (?stack, \text{None})$
using *step-correct valid-def* **by** *simp*
with *assms(3)* **have** $\text{nonterminating } (?stack, \text{None})$
by *simp*
then show *?thesis*
using *r-result-nonempty-stack assms(1,2)* **by** *simp*
qed

Now we can define the universal partial recursive function. This function executes *r-result*

for increasing time bounds, waits for it to reach a final configuration, and then extracts its result value. If no final configuration is reached, the universal function diverges.

definition *r-univ* \equiv

Cn 2 r-dec [Cn 2 r-result [Mn 2 (Cn 3 r-not [r-result]), Id 2 0, Id 2 1]]

lemma *r-univ-recfn [simp]: recfn 2 r-univ*

unfolding *r-univ-def* **by** *simp*

theorem *r-univ:*

assumes *i = encode f and recfn (e-length x) f*

shows *eval r-univ [i, x] = eval f (list-decode x)*

proof –

let *?cond = Cn 3 r-not [r-result]*

let *?while = Mn 2 ?cond*

let *?res = Cn 2 r-result [?while, Id 2 0, Id 2 1]*

let *?xs = list-decode x*

have ***: *eval ?cond [t, i, x] \downarrow = (if eval r-result [t, i, x] \downarrow = 0 then 1 else 0)* **for** *t*

proof –

have *eval ?cond [t, i, x] = eval r-not [the (eval r-result [t, i, x])]*

using *r-result-total* **by** *simp*

moreover **have** *eval r-result [t, i, x] \downarrow*

by (*simp add: r-result-total*)

ultimately show *?thesis* **by** *auto*

qed

show *?thesis*

proof (*cases eval f ?xs \uparrow*)

case *True*

then show *?thesis*

unfolding *r-univ-def* **using** ** r-result-diverg[OF assms] eval-Mn-diverg* **by** *simp*

next

case *False*

then obtain *v* **where** *v: eval f ?xs \downarrow = v* **by** *auto*

then obtain *t* **where** *t:*

$\forall t' \geq t. \text{eval } r\text{-result } [t', i, x] \downarrow = \text{Suc } v$

$\forall t' < t. \text{eval } r\text{-result } [t', i, x] \downarrow = 0$

using *r-result-converg[OF assms]* **by** *blast*

then have

$\forall t' \geq t. \text{eval } ?\text{cond } [t', i, x] \downarrow = 0$

$\forall t' < t. \text{eval } ?\text{cond } [t', i, x] \downarrow = 1$

using *** **by** *simp-all*

then have *eval ?while [i, x] \downarrow = t*

using *eval-Mn-convergI[of 2 ?cond [i, x] t]* **by** *simp*

then have *eval ?res [i, x] = eval r-result [t, i, x]*

by *simp*

then have *eval ?res [i, x] \downarrow = Suc v*

using *t(1)* **by** *simp*

then show *?thesis*

unfolding *r-univ-def* **using** *v* **by** *simp*

qed

qed

theorem *r-univ':*

assumes *recfn (e-length x) f*

shows *eval r-univ [encode f, x] = eval f (list-decode x)*

using *r-univ assms* **by** *simp*

Universal functions for every arity can be built from *r-univ*.

definition *r-universal* :: *nat* \Rightarrow *recf* **where**
r-universal *n* \equiv *Cn* (*Suc* *n*) *r-univ* [*Id* (*Suc* *n*) 0, *r-shift* (*r-list-encode* (*n* - 1))]

lemma *r-universal-recfn* [*simp*]: *n* > 0 \implies *recfn* (*Suc* *n*) (*r-universal* *n*)
unfolding *r-universal-def* **by** *simp*

lemma *r-universal*:
assumes *recfn* *n* *f* **and** *length* *xs* = *n*
shows *eval* (*r-universal* *n*) (*encode* *f* # *xs*) = *eval* *f* *xs*
unfolding *r-universal-def* **using** *welff-arity-nonzero* *assms* *r-list-encode* *r-univ'*
by *fastforce*

We will mostly be concerned with computing unary functions. Hence we introduce separate functions for this case.

definition *r-result1* \equiv
Cn 3 *r-result* [*Id* 3 0, *Id* 3 1, *Cn* 3 *r-singleton-encode* [*Id* 3 2]]

lemma *r-result1-prim* [*simp*]: *prim-recfn* 3 *r-result1*
unfolding *r-result1-def* **by** *simp*

lemma *r-result1-total*: *total* *r-result1*
using *Mn-free-imp-total* **by** *simp*

lemma *r-result1* [*simp*]:
eval *r-result1* [*t*, *i*, *x*] = *eval* *r-result* [*t*, *i*, *singleton-encode* *x*]
unfolding *r-result1-def* **by** *simp*

The following function will be our standard Gödel numbering of all unary partial recursive functions.

definition *r-phi* \equiv *r-universal* 1

lemma *r-phi-recfn* [*simp*]: *recfn* 2 *r-phi*
unfolding *r-phi-def* **by** *simp*

theorem *r-phi*:
assumes *i* = *encode* *f* **and** *recfn* 1 *f*
shows *eval* *r-phi* [*i*, *x*] = *eval* *f* [*x*]
unfolding *r-phi-def* **using** *r-universal* *assms* **by** *force*

corollary *r-phi'*:
assumes *recfn* 1 *f*
shows *eval* *r-phi* [*encode* *f*, *x*] = *eval* *f* [*x*]
using *assms* *r-phi* **by** *simp*

lemma *r-phi''*: *eval* *r-phi* [*i*, *x*] = *eval* *r-univ* [*i*, *singleton-encode* *x*]
unfolding *r-universal-def* *r-phi-def* **using** *r-list-encode* **by** *simp*

1.6 Applications of the universal function

In this section we shall see some ways *r-univ* and *r-result* can be used.

1.6.1 Lazy conditional evaluation

With the help of *r-univ* we can now define a lazy variant of *r-ifz*, in which only one branch is evaluated.

definition *r-lazyifzero* :: *nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *recf* **where**

r-lazyifzero *n j₁ j₂* \equiv
Cn (*Suc* (*Suc* *n*)) *r-univ*
 [*Cn* (*Suc* (*Suc* *n*)) *r-ifz* [*Id* (*Suc* (*Suc* *n*)) 0, *r-constn* (*Suc* *n*) *j₁*, *r-constn* (*Suc* *n*) *j₂*],
r-shift (*r-list-encode* *n*)]

lemma *r-lazyifzero-recfn*: *recfn* (*Suc* (*Suc* *n*)) (*r-lazyifzero* *n j₁ j₂*)
using *r-lazyifzero-def* **by** *simp*

lemma *r-lazyifzero*:

assumes *length xs = Suc n*

and *j₁ = encode f₁*

and *j₂ = encode f₂*

and *recfn* (*Suc* *n*) *f₁*

and *recfn* (*Suc* *n*) *f₂*

shows *eval* (*r-lazyifzero* *n j₁ j₂*) (*c # xs*) = (*if* *c = 0* *then eval f₁ xs* *else eval f₂ xs*)

proof –

let *?a = r-constn* (*Suc* *n*) *n*

let *?b = Cn* (*Suc* (*Suc* *n*)) *r-ifz*

[*Id* (*Suc* (*Suc* *n*)) 0, *r-constn* (*Suc* *n*) *j₁*, *r-constn* (*Suc* *n*) *j₂*]

let *?c = r-shift* (*r-list-encode* *n*)

have *eval ?a* (*c # xs*) $\downarrow = n$

using *assms(1)* **by** *simp*

moreover have *eval ?b* (*c # xs*) $\downarrow =$ (*if* *c = 0* *then j₁* *else j₂*)

using *assms(1)* **by** *simp*

moreover have *eval ?c* (*c # xs*) $\downarrow =$ *list-encode xs*

using *assms(1)* *r-list-encode* *r-shift* **by** *simp*

ultimately have *eval* (*r-lazyifzero* *n j₁ j₂*) (*c # xs*) =

eval r-univ [*if* *c = 0* *then j₁* *else j₂*, *list-encode xs*]

unfolding *r-lazyifzero-def* **using** *r-lazyifzero-recfn* *assms(1)* **by** *simp*

then show *?thesis* **using** *assms* *r-univ* **by** *simp*

qed

definition *r-lifz* :: *recf* \Rightarrow *recf* \Rightarrow *recf* **where**

r-lifz *f g* \equiv *r-lazyifzero* (*arity* *f* – 1) (*encode* *f*) (*encode* *g*)

lemma *r-lifz-recfn* [*simp*]:

assumes *recfn* *n f* **and** *recfn* *n g*

shows *recfn* (*Suc* *n*) (*r-lifz* *f g*)

using *assms* *r-lazyifzero-recfn* *r-lifz-def* *wellf-arity-nonzero* **by** *auto*

lemma *r-lifz* [*simp*]:

assumes *length xs = n* **and** *recfn* *n f* **and** *recfn* *n g*

shows *eval* (*r-lifz* *f g*) (*c # xs*) = (*if* *c = 0* *then eval f xs* *else eval g xs*)

using *assms* *r-lazyifzero* *r-lifz-def* *wellf-arity-nonzero*

by (*metis* *One-nat-def* *Suc-pred*)

1.6.2 Enumerating the domains of partial recursive functions

In this section we define a binary function *enumdom* such that for all *i*, the domain of φ_i equals $\{\text{enumdom}(i, x) \mid \text{enumdom}(i, x) \downarrow\}$. In other words, the image of *enumdom_i*

is the domain of φ_i .

First we need some more properties of *r-leap* and *r-result*.

lemma *r-leap-Suc*: $eval\ r\ leap\ [Suc\ t,\ i,\ x] = eval\ r\ step\ [the\ (eval\ r\ leap\ [t,\ i,\ x])]$

proof –

have $eval\ r\ leap\ [Suc\ t,\ i,\ x] =$
 $eval\ (Cn\ 4\ r\ step\ [Id\ 4\ 1])\ [t,\ the\ (eval\ r\ leap\ [t,\ i,\ x]),\ i,\ x]$
using *r-leap-total eval-Pr-converg-Suc r-leap-def*
by (*metis length-Cons list.size(3) numeral-2-eq-2 numeral-3-eq-3 r-leap-prim*)
then show *?thesis using r-step-prim by auto*

qed

lemma *r-leap-Suc-saturating*:

assumes $pdec1\ (the\ (eval\ r\ leap\ [t,\ i,\ x])) = 0$
shows $eval\ r\ leap\ [Suc\ t,\ i,\ x] = eval\ r\ leap\ [t,\ i,\ x]$

proof –

let $?e = eval\ r\ leap\ [t,\ i,\ x]$
have $eval\ r\ step\ [the\ ?e] \downarrow = estep\ (the\ ?e)$
using *r-step by simp*
then have $eval\ r\ step\ [the\ ?e] \downarrow = prod\ encode\ (0,\ e2rv\ (the\ ?e))$
using *estep-def assms by (simp add: e2stack-def)*
then have $eval\ r\ step\ [the\ ?e] \downarrow = prod\ encode\ (pdec1\ (the\ ?e),\ pdec2\ (the\ ?e))$
using *assms by (simp add: e2rv-def)*
then have $eval\ r\ step\ [the\ ?e] \downarrow = the\ ?e$ **by** *simp*
then show *?thesis using r-leap-total r-leap-Suc by simp*

qed

lemma *r-result-Suc-saturating*:

assumes $eval\ r\ result\ [t,\ i,\ x] \downarrow = Suc\ v$
shows $eval\ r\ result\ [Suc\ t,\ i,\ x] \downarrow = Suc\ v$

proof –

let $?r = \lambda t.\ eval\ r\ ifz\ [pdec1\ (the\ (eval\ r\ leap\ [t,\ i,\ x])),\ pdec2\ (the\ (eval\ r\ leap\ [t,\ i,\ x])),\ 0]$
have $?r\ t \downarrow = Suc\ v$
using *assms unfolding r-result-def using r-leap-total r-leap-prim by simp*
then have $pdec1\ (the\ (eval\ r\ leap\ [t,\ i,\ x])) = 0$
using *option.sel by fastforce*
then have $eval\ r\ leap\ [Suc\ t,\ i,\ x] = eval\ r\ leap\ [t,\ i,\ x]$
using *r-leap-Suc-saturating by simp*
moreover have $eval\ r\ result\ [t,\ i,\ x] = ?r\ t$
unfolding *r-result-def using r-leap-total r-leap-prim by simp*
moreover have $eval\ r\ result\ [Suc\ t,\ i,\ x] = ?r\ (Suc\ t)$
unfolding *r-result-def using r-leap-total r-leap-prim by simp*
ultimately have $eval\ r\ result\ [Suc\ t,\ i,\ x] = eval\ r\ result\ [t,\ i,\ x]$
by *simp*
with *assms show ?thesis by simp*

qed

lemma *r-result-saturating*:

assumes $eval\ r\ result\ [t,\ i,\ x] \downarrow = Suc\ v$
shows $eval\ r\ result\ [t + d,\ i,\ x] \downarrow = Suc\ v$
using *r-result-Suc-saturating assms by (induction d) simp-all*

lemma *r-result-converg'*:

assumes $eval\ r\ univ\ [i,\ x] \downarrow = v$
shows $\exists t.\ (\forall t' \geq t.\ eval\ r\ result\ [t',\ i,\ x] \downarrow = Suc\ v) \wedge (\forall t' < t.\ eval\ r\ result\ [t',\ i,\ x] \downarrow = 0)$

proof –

let $?f = Cn\ 3\ r\text{-not}\ [r\text{-result}]$
let $?m = Mn\ 2\ ?f$
have $recfn\ 2\ ?m$ **by** *simp*
have $eval\text{-}m$: $eval\ ?m\ [i, x] \downarrow$
proof
 assume $eval\ ?m\ [i, x] \uparrow$
 then have $eval\ r\text{-univ}\ [i, x] \uparrow$
 unfolding $r\text{-univ}\text{-def}$ **by** *simp*
 with *assms* **show** *False* **by** *simp*
qed
then obtain t **where** t : $eval\ ?m\ [i, x] \downarrow = t$
 by *auto*
then have $f\text{-}t$: $eval\ ?f\ [t, i, x] \downarrow = 0$ **and** $f\text{-less}\text{-}t$: $\bigwedge y. y < t \implies eval\ ?f\ [y, i, x] \downarrow \neq 0$
 using $eval\text{-}Mn\text{-convergE}$ [of 2 $?f\ [i, x]\ t$] $\langle recfn\ 2\ ?m \rangle$
 by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-1 length-Cons list.size(3)*)
have $eval\text{-}Cn2$: $eval\ (Cn\ 2\ r\text{-result}\ [?m, Id\ 2\ 0, Id\ 2\ 1])\ [i, x] \downarrow$
proof
 assume $eval\ (Cn\ 2\ r\text{-result}\ [?m, Id\ 2\ 0, Id\ 2\ 1])\ [i, x] \uparrow$
 then have $eval\ r\text{-univ}\ [i, x] \uparrow$
 unfolding $r\text{-univ}\text{-def}$ **by** *simp*
 with *assms* **show** *False* **by** *simp*
qed
have $eval\ r\text{-result}\ [t, i, x] \downarrow = Suc\ v$
proof (*rule ccontr*)
 assume $neg\text{-}Suc$: $\neg eval\ r\text{-result}\ [t, i, x] \downarrow = Suc\ v$
 show *False*
 proof (*cases eval r-result [t, i, x] = None*)
 case *True*
 then show $?thesis$ **using** $f\text{-}t$ **by** *simp*
 next
 case *False*
 then obtain w **where** w : $eval\ r\text{-result}\ [t, i, x] \downarrow = w\ w \neq Suc\ v$
 using $neg\text{-}Suc$ **by** *auto*
 moreover have $eval\ r\text{-result}\ [t, i, x] \downarrow \neq 0$
 by (*rule ccontr*; *use f-t in auto*)
 ultimately have $w \neq 0$ **by** *simp*
 have $eval\ (Cn\ 2\ r\text{-result}\ [?m, Id\ 2\ 0, Id\ 2\ 1])\ [i, x] =$
 $eval\ r\text{-result}\ [the\ (eval\ ?m\ [i, x]), i, x]$
 using $eval\text{-}m$ **by** *simp*
 with $w\ t$ **have** $eval\ (Cn\ 2\ r\text{-result}\ [?m, Id\ 2\ 0, Id\ 2\ 1])\ [i, x] \downarrow = w$
 by *simp*
 moreover have $eval\ r\text{-univ}\ [i, x] =$
 $eval\ r\text{-dec}\ [the\ (eval\ (Cn\ 2\ r\text{-result}\ [?m, Id\ 2\ 0, Id\ 2\ 1])\ [i, x])]$
 unfolding $r\text{-univ}\text{-def}$ **using** $eval\text{-}Cn2$ **by** *simp*
 ultimately have $eval\ r\text{-univ}\ [i, x] = eval\ r\text{-dec}\ [w]$ **by** *simp*
 then have $eval\ r\text{-univ}\ [i, x] \downarrow = w - 1$ **by** *simp*
 with *assms* $\langle w \neq 0 \rangle\ w$ **show** $?thesis$ **by** *simp*
qed
qed
then have $\forall t' \geq t. eval\ r\text{-result}\ [t', i, x] \downarrow = Suc\ v$
 using $r\text{-result}\text{-saturating}\ le\text{-}Suc\text{-ex}$ **by** *blast*
moreover have $eval\ r\text{-result}\ [y, i, x] \downarrow = 0$ **if** $y < t$ **for** y
proof (*rule ccontr*)
 assume $neg0$: $eval\ r\text{-result}\ [y, i, x] \neq Some\ 0$
 then show *False*
 proof (*cases eval r-result [y, i, x] = None*)

```

    case True
    then show ?thesis using f-less-t ⟨y < t⟩ by fastforce
next
case False
then obtain v where eval r-result [y, i, x] ↓= v v ≠ 0
  using neq0 by auto
then have eval ?f [y, i, x] ↓= 0 by simp
then show ?thesis using f-less-t ⟨y < t⟩ by simp
qed
qed
ultimately show ?thesis by auto
qed

lemma r-result-diverg':
  assumes eval r-univ [i, x] ↑
  shows eval r-result [t, i, x] ↓= 0
proof (rule ccontr)
  let ?f = Cn 3 r-not [r-result]
  let ?m = Mn 2 ?f
  assume eval r-result [t, i, x] ≠ Some 0
  with r-result-total have eval r-result [t, i, x] ↓≠ 0 by simp
  then have eval ?f [t, i, x] ↓= 0 by auto
  moreover have eval ?f [y, i, x] ↓ if y < t for y
    using r-result-total by simp
  ultimately have ∃ z. eval ?f (z # [i, x]) ↓= 0 ∧ (∀ y < z. eval ?f (y # [i, x]) ↓)
    by blast
  then have eval ?m [i, x] ↓ by simp
  then have eval r-univ [i, x] ↓
    unfolding r-univ-def using r-result-total by simp
  with assms show False by simp
qed

lemma r-result-bivalent':
  assumes eval r-univ [i, x] ↓= v
  shows eval r-result [t, i, x] ↓= Suc v ∨ eval r-result [t, i, x] ↓= 0
  using r-result-converg'[OF assms] not-less by blast

lemma r-result-Some':
  assumes eval r-result [t, i, x] ↓= Suc v
  shows eval r-univ [i, x] ↓= v
proof (rule ccontr)
  assume not-v: ¬ eval r-univ [i, x] ↓= v
  show False
proof (cases eval r-univ [i, x] ↑)
  case True
  then show ?thesis
    using assms r-result-diverg' by simp
next
  case False
  then obtain w where w: eval r-univ [i, x] ↓= w w ≠ v
    using not-v by auto
  then have eval r-result [t, i, x] ↓= Suc w ∨ eval r-result [t, i, x] ↓= 0
    using r-result-bivalent' by simp
  then show ?thesis using assms not-v w by simp
qed
qed

```

lemma *r-result1-converg'*:
assumes *eval r-phi* $[i, x] \downarrow = v$
shows $\exists t.$
 $(\forall t' \geq t. \text{eval } r\text{-result1 } [t', i, x] \downarrow = \text{Suc } v) \wedge$
 $(\forall t' < t. \text{eval } r\text{-result1 } [t', i, x] \downarrow = 0)$
using *assms r-result1 r-result-converg' r-phi''* **by** *simp*

lemma *r-result1-diverg'*:
assumes *eval r-phi* $[i, x] \uparrow$
shows *eval r-result1* $[t, i, x] \downarrow = 0$
using *assms r-result1 r-result-diverg' r-phi''* **by** *simp*

lemma *r-result1-Some'*:
assumes *eval r-result1* $[t, i, x] \downarrow = \text{Suc } v$
shows *eval r-phi* $[i, x] \downarrow = v$
using *assms r-result1 r-result-Some' r-phi''* **by** *simp*

The next function performs dovetailing in order to evaluate φ_i for every argument for arbitrarily many steps. Given i and z , the function decodes z into a pair (x, t) and outputs zero (meaning “true”) iff. the computation of φ_i on input x halts after at most t steps. Fixing i and varying z will eventually compute φ_i for every argument in the domain of φ_i sufficiently long for it to converge.

definition *r-dovetail* \equiv
 $Cn\ 2\ r\text{-not } [Cn\ 2\ r\text{-result1 } [Cn\ 2\ r\text{-pdec2 } [Id\ 2\ 1], Id\ 2\ 0, Cn\ 2\ r\text{-pdec1 } [Id\ 2\ 1]]]$

lemma *r-dovetail-prim*: *prim-recfn 2 r-dovetail*
by (*simp add: r-dovetail-def*)

lemma *r-dovetail*:
eval r-dovetail $[i, z] \downarrow =$
 $(\text{if the } (\text{eval } r\text{-result1 } [pdec2\ z, i, pdec1\ z]) > 0 \text{ then } 0 \text{ else } 1)$
unfolding *r-dovetail-def* **using** *r-result-total* **by** *simp*

The function *enumdom* works as follows in order to enumerate exactly the domain of φ_i . Given i and y it searches for the minimum $z \geq y$ for which the dovetail function returns true. This z is decoded into (x, t) and the x is output. In this way every value output by *enumdom* is in the domain of φ_i by construction of *r-dovetail*. Conversely an x in the domain will be output for $y = (x, t)$ where t is such that φ_i halts on x within t steps.

definition *r-dovedelay* \equiv
 $Cn\ 3\ r\text{-and}$
 $[Cn\ 3\ r\text{-dovetail } [Id\ 3\ 1, Id\ 3\ 0],$
 $Cn\ 3\ r\text{-ifle } [Id\ 3\ 2, Id\ 3\ 0, r\text{-constn } 2\ 0, r\text{-constn } 2\ 1]]$

lemma *r-dovedelay-prim*: *prim-recfn 3 r-dovedelay*
unfolding *r-dovedelay-def* **using** *r-dovetail-prim* **by** *simp*

lemma *r-dovedelay*:
eval r-dovedelay $[z, i, y] \downarrow =$
 $(\text{if the } (\text{eval } r\text{-result1 } [pdec2\ z, i, pdec1\ z]) > 0 \wedge y \leq z \text{ then } 0 \text{ else } 1)$
by (*simp add: r-dovedelay-def r-dovetail r-dovetail-prim*)

definition *r-enumdom* $\equiv Cn\ 2\ r\text{-pdec1 } [Mn\ 2\ r\text{-dovedelay}]$

lemma *r-enumdom-recfn* [*simp*]: *recfn* 2 *r-enumdom*
by (*simp add: r-enumdom-def r-dovedelay-prim*)

lemma *r-enumdom* [*simp*]:
eval r-enumdom [*i*, *y*] =
 (if $\exists z. \text{eval } r\text{-dovedelay } [z, i, y] \downarrow = 0$
 then *Some* (*pdec1* (*LEAST* $z. \text{eval } r\text{-dovedelay } [z, i, y] \downarrow = 0$))
 else *None*)

proof –
let *?h* = *Mn* 2 *r-dovedelay*
have *total r-dovedelay*
using *r-dovedelay-prim* **by** *blast*
then have *eval ?h* [*i*, *y*] =
 (if ($\exists z. \text{eval } r\text{-dovedelay } [z, i, y] \downarrow = 0$)
 then *Some* (*LEAST* $z. \text{eval } r\text{-dovedelay } [z, i, y] \downarrow = 0$)
 else *None*)
using *r-dovedelay-prim r-enumdom-recfn eval-Mn-convergI* **by** *simp*
then show *?thesis*
unfolding *r-enumdom-def* **using** *r-dovedelay-prim* **by** *simp*
qed

If *i* is the code of the empty function, *r-enumdom* has an empty domain, too.

lemma *r-enumdom-empty-domain*:
assumes $\bigwedge x. \text{eval } r\text{-phi } [i, x] \uparrow$
shows $\bigwedge y. \text{eval } r\text{-enumdom } [i, y] \uparrow$
using *assms r-result1-diverg' r-dovedelay* **by** *simp*

If *i* is the code of a function with non-empty domain, *r-enumdom* enumerates its domain.

lemma *r-enumdom-nonempty-domain*:
assumes *eval r-phi* [*i*, *x*₀] \downarrow
shows $\bigwedge y. \text{eval } r\text{-enumdom } [i, y] \downarrow$
and $\bigwedge x. \text{eval } r\text{-phi } [i, x] \downarrow \longleftrightarrow (\exists y. \text{eval } r\text{-enumdom } [i, y] \downarrow = x)$

proof –
show *eval r-enumdom* [*i*, *y*] \downarrow **for** *y*
proof –
obtain *t* **where** $t: \forall t' \geq t. \text{the } (\text{eval } r\text{-result1 } [t', i, x_0]) > 0$
using *assms r-result1-converg'* **by** *fastforce*
let *?z* = *prod-encode* (*x*₀, *max* *t* *y*)
have $y \leq ?z$
using *le-prod-encode-2 max.bounded-iff* **by** *blast*
moreover have *pdec2* *?z* $\geq t$ **by** *simp*
ultimately have *the* (*eval r-result1* [*pdec2* *?z*, *i*, *pdec1* *?z*]) > 0
using *t* **by** *simp*
with $\langle y \leq ?z \rangle$ *r-dovedelay* **have** *eval r-dovedelay* [*?z*, *i*, *y*] $\downarrow = 0$
by *presburger*
then show *eval r-enumdom* [*i*, *y*] \downarrow
using *r-enumdom* **by** *auto*

qed
show *eval r-phi* [*i*, *x*] $\downarrow = (\exists y. \text{eval } r\text{-enumdom } [i, y] \downarrow = x)$ **for** *x*

proof
show $\exists y. \text{eval } r\text{-enumdom } [i, y] \downarrow = x$ **if** *eval r-phi* [*i*, *x*] \downarrow **for** *x*
proof –
from *that* **obtain** *v* **where** *eval r-phi* [*i*, *x*] $\downarrow = v$ **by** *auto*
then obtain *t* **where** $t: \text{the } (\text{eval } r\text{-result1 } [t, i, x]) > 0$
using *r-result1-converg'* *assms*

```

    by (metis Zero-not-Suc dual-order.refl option.sel zero-less-iff-neq-zero)
  let ?y = prod-encode (x, t)
  have eval r-dovedelay [?y, i, ?y] ↓= 0
    using r-dovedelay t by simp
  moreover from this have (LEAST z. eval r-dovedelay [z, i, ?y] ↓= 0) = ?y
    using gr-implies-not-zero r-dovedelay by (intro Least-equality; fastforce)
  ultimately have eval r-enumdom [i, ?y] ↓= x
    using r-enumdom by auto
  then show ?thesis by blast
qed
show eval r-phi [i, x] ↓ if ∃ y. eval r-enumdom [i, y] ↓= x for x
proof -
  from that obtain y where y: eval r-enumdom [i, y] ↓= x
    by auto
  then have eval r-enumdom [i, y] ↓
    by simp
  then have
    ∃ z. eval r-dovedelay [z, i, y] ↓= 0 and
    *: eval r-enumdom [i, y] ↓= pdec1 (LEAST z. eval r-dovedelay [z, i, y] ↓= 0)
    (is - ↓= pdec1 ?z)
    using r-enumdom by metis+
  then have z: eval r-dovedelay [?z, i, y] ↓= 0
    by (meson wellorder-Least-lemma(1))
  have the (eval r-result1 [pdec2 ?z, i, pdec1 ?z]) > 0
  proof (rule ccontr)
    assume ¬ (the (eval r-result1 [pdec2 ?z, i, pdec1 ?z]) > 0)
    then show False
      using r-dovedelay z by simp
  qed
  then have eval r-phi [i, pdec1 ?z] ↓
    using r-result1-diverg' assms by fastforce
  then show ?thesis using y * by auto
qed
qed
qed

```

For every φ_i with non-empty domain there is a total recursive function that enumerates the domain of φ_i .

lemma *nonempty-domain-enumerable*:

```

  assumes eval r-phi [i, x0] ↓
  shows ∃ g. recfn 1 g ∧ total g ∧ (∀ x. eval r-phi [i, x] ↓ ↔ (∃ y. eval g [y] ↓= x))
proof -
  define g where g ≡ Cn 1 r-enumdom [r-const i, Id 1 0]
  then have recfn 1 g by simp
  moreover from this have total g
    using totalI1[of g] g-def assms r-enumdom-nonempty-domain(1) by simp
  moreover have eval r-phi [i, x] ↓ ↔ (∃ y. eval g [y] ↓= x) for x
    unfolding g-def using r-enumdom-nonempty-domain(2)[OF assms] by simp
  ultimately show ?thesis by auto
qed

```

1.6.3 Concurrent evaluation of functions

We define a function that simulates two *recfs* “concurrently” for the same argument and returns the result of the one converging first. If both diverge, so does the simulation

function.

definition *r-both* \equiv

```

Cn 4 r-ifz
[Cn 4 r-result1 [Id 4 0, Id 4 1, Id 4 3],
Cn 4 r-ifz
[Cn 4 r-result1 [Id 4 0, Id 4 2, Id 4 3],
Cn 4 r-prod-encode [r-constn 3 2, r-constn 3 0],
Cn 4 r-prod-encode
[r-constn 3 1, Cn 4 r-dec [Cn 4 r-result1 [Id 4 0, Id 4 2, Id 4 3]]],
Cn 4 r-prod-encode
[r-constn 3 0, Cn 4 r-dec [Cn 4 r-result1 [Id 4 0, Id 4 1, Id 4 3]]]]

```

lemma *r-both-prim* [*simp*]: *prim-recfn* 4 *r-both*
unfolding *r-both-def* **by** *simp*

lemma *r-both*:

```

assumes  $\bigwedge x. \text{eval } r\text{-phi } [i, x] = \text{eval } f [x]$ 
and  $\bigwedge x. \text{eval } r\text{-phi } [j, x] = \text{eval } g [x]$ 
shows  $\text{eval } f [x] \uparrow \wedge \text{eval } g [x] \uparrow \implies \text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (2, 0)$ 
and  $\llbracket \text{eval } r\text{-result1 } [t, i, x] \downarrow = 0; \text{eval } r\text{-result1 } [t, j, x] \downarrow = 0 \rrbracket \implies$ 
 $\text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (2, 0)$ 
and  $\text{eval } r\text{-result1 } [t, i, x] \downarrow = \text{Suc } v \implies$ 
 $\text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } f [x]))$ 
and  $\llbracket \text{eval } r\text{-result1 } [t, i, x] \downarrow = 0; \text{eval } r\text{-result1 } [t, j, x] \downarrow = \text{Suc } v \rrbracket \implies$ 
 $\text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (1, \text{the } (\text{eval } g [x]))$ 

```

proof –

```

have r-result-total [simp]:  $\text{eval } r\text{-result } [t, k, x] \downarrow$  for t k x
using r-result-total by simp
{
assume  $\text{eval } f [x] \uparrow \wedge \text{eval } g [x] \uparrow$ 
then have  $\text{eval } r\text{-result1 } [t, i, x] \downarrow = 0$  and  $\text{eval } r\text{-result1 } [t, j, x] \downarrow = 0$ 
using assms r-result1-diverg' by auto
then show  $\text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (2, 0)$ 
unfolding r-both-def by simp
next
assume  $\text{eval } r\text{-result1 } [t, i, x] \downarrow = 0$  and  $\text{eval } r\text{-result1 } [t, j, x] \downarrow = 0$ 
then show  $\text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (2, 0)$ 
unfolding r-both-def by simp
next
assume  $\text{eval } r\text{-result1 } [t, i, x] \downarrow = \text{Suc } v$ 
moreover from this have  $\text{eval } r\text{-result1 } [t, i, x] \downarrow = \text{Suc } (\text{the } (\text{eval } f [x]))$ 
using assms r-result1-Some' by fastforce
ultimately show  $\text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } f [x]))$ 
unfolding r-both-def by auto
next
assume  $\text{eval } r\text{-result1 } [t, i, x] \downarrow = 0$  and  $\text{eval } r\text{-result1 } [t, j, x] \downarrow = \text{Suc } v$ 
moreover from this have  $\text{eval } r\text{-result1 } [t, j, x] \downarrow = \text{Suc } (\text{the } (\text{eval } g [x]))$ 
using assms r-result1-Some' by fastforce
ultimately show  $\text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (1, \text{the } (\text{eval } g [x]))$ 
unfolding r-both-def by auto
}
qed

```

definition *r-parallel* \equiv

```

Cn 3 r-both [Mn 3 (Cn 4 r-le [Cn 4 r-pdec1 [r-both], r-constn 3 1]), Id 3 0, Id 3 1, Id 3 2]

```

lemma *r-parallel-recfn [simp]*: *recfn 3 r-parallel*
unfolding *r-parallel-def* **by** *simp*

lemma *r-parallel*:

assumes $\bigwedge x. \text{eval } r\text{-phi } [i, x] = \text{eval } f [x]$
and $\bigwedge x. \text{eval } r\text{-phi } [j, x] = \text{eval } g [x]$
shows $\text{eval } f [x] \uparrow \wedge \text{eval } g [x] \uparrow \implies \text{eval } r\text{-parallel } [i, j, x] \uparrow$
and $\text{eval } f [x] \downarrow \wedge \text{eval } g [x] \uparrow \implies$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } f [x]))$
and $\text{eval } g [x] \downarrow \wedge \text{eval } f [x] \uparrow \implies$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (1, \text{the } (\text{eval } g [x]))$
and $\text{eval } f [x] \downarrow \wedge \text{eval } g [x] \downarrow \implies$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } f [x])) \vee$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (1, \text{the } (\text{eval } g [x]))$

proof –

let $?cond = Cn \ 4 \ r\text{-le } [Cn \ 4 \ r\text{-pdec1 } [r\text{-both}], r\text{-constn } 3 \ 1]$
define m **where** $m = Mn \ 3 \ ?cond$
then have $m: r\text{-parallel} = Cn \ 3 \ r\text{-both } [m, Id \ 3 \ 0, Id \ 3 \ 1, Id \ 3 \ 2]$
unfolding *r-parallel-def* **by** *simp*
from $m\text{-def}$ **have** *recfn 3 m* **by** *simp*
{
assume $\text{eval } f [x] \uparrow \wedge \text{eval } g [x] \uparrow$
then have $\forall t. \text{eval } r\text{-both } [t, i, j, x] \downarrow = \text{prod-encode } (2, 0)$
using *assms r-both* **by** *simp*
then have $\text{eval } ?cond [t, i, j, x] \downarrow = 1$ **for** t
by *simp*
then have $\text{eval } m [i, j, x] \uparrow$
unfolding $m\text{-def}$ **using** *eval-Mn-diverg* **by** *simp*
then have $\text{eval } (Cn \ 3 \ r\text{-both } [m, Id \ 3 \ 0, Id \ 3 \ 1, Id \ 3 \ 2]) [i, j, x] \uparrow$
using $\langle \text{recfn } 3 \ m \rangle$ **by** *simp*
then show $\text{eval } r\text{-parallel } [i, j, x] \uparrow$
using m **by** *simp*
next
assume $\text{eval } f [x] \downarrow \wedge \text{eval } g [x] \downarrow$
then obtain $vf \ vg$ **where** $v: \text{eval } f [x] \downarrow = vf \ \text{eval } g [x] \downarrow = vg$
by *auto*
then obtain tf **where** tf :
 $\forall t \geq tf. \text{eval } r\text{-result1 } [t, i, x] \downarrow = \text{Suc } vf$
 $\forall t < tf. \text{eval } r\text{-result1 } [t, i, x] \downarrow = 0$
using *r-result1-converg'* *assms* **by** *metis*
from v **obtain** tg **where** tg :
 $\forall t \geq tg. \text{eval } r\text{-result1 } [t, j, x] \downarrow = \text{Suc } vg$
 $\forall t < tg. \text{eval } r\text{-result1 } [t, j, x] \downarrow = 0$
using *r-result1-converg'* *assms* **by** *metis*
show $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } f [x])) \vee$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (1, \text{the } (\text{eval } g [x]))$
proof (*cases* $tf \leq tg$)
case *True*
with $tg(2)$ **have** $j0: \forall t < tf. \text{eval } r\text{-result1 } [t, j, x] \downarrow = 0$
by *simp*
have $*$: $\text{eval } r\text{-both } [tf, i, j, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } f [x]))$
using *r-both(3)* *assms* $tf(1)$ **by** *simp*
have $\text{eval } m [i, j, x] \downarrow = tf$
unfolding $m\text{-def}$
proof (*rule* *eval-Mn-convergI*)

```

show recfn (length [i, j, x]) (Mn 3 ?cond) by simp
have eval (Cn 4 r-pdec1 [r-both]) [tf, i, j, x]  $\downarrow = 0$ 
  using * by simp
then show eval ?cond [tf, i, j, x]  $\downarrow = 0$  by simp
have eval r-both [t, i, j, x]  $\downarrow = \text{prod-encode } (2, 0)$  if  $t < tf$  for  $t$ 
  using tf(2) r-both(2) assms that j0 by simp
then have eval ?cond [t, i, j, x]  $\downarrow = 1$  if  $t < tf$  for  $t$ 
  using that by simp
then show  $\bigwedge y. y < tf \implies \text{eval } ?\text{cond } [y, i, j, x] \downarrow \neq 0$  by simp
qed
moreover have eval r-parallel [i, j, x] =
  eval (Cn 3 r-both [m, Id 3 0, Id 3 1, Id 3 2]) [i, j, x]
  using m by simp
ultimately have eval r-parallel [i, j, x] = eval r-both [tf, i, j, x]
  using  $\langle \text{recfn } 3 \ m \rangle$  by simp
with * have eval r-parallel [i, j, x]  $\downarrow = \text{prod-encode } (0, \text{the } (\text{eval } f [x]))$ 
  by simp
then show ?thesis by simp
next
case False
with tf(2) have i0:  $\forall t \leq tg. \text{eval } r\text{-result1 } [t, i, x] \downarrow = 0$ 
  by simp
then have *: eval r-both [tg, i, j, x]  $\downarrow = \text{prod-encode } (1, \text{the } (\text{eval } g [x]))$ 
  using assms r-both(4) tg(1) by auto
have eval m [i, j, x]  $\downarrow = tg$ 
  unfolding m-def
proof (rule eval-Mn-convergI)
  show recfn (length [i, j, x]) (Mn 3 ?cond) by simp
  have eval (Cn 4 r-pdec1 [r-both]) [tg, i, j, x]  $\downarrow = 1$ 
    using * by simp
  then show eval ?cond [tg, i, j, x]  $\downarrow = 0$  by simp
  have eval r-both [t, i, j, x]  $\downarrow = \text{prod-encode } (2, 0)$  if  $t < tg$  for  $t$ 
    using tg(2) r-both(2) assms that i0 by simp
  then have eval ?cond [t, i, j, x]  $\downarrow = 1$  if  $t < tg$  for  $t$ 
    using that by simp
  then show  $\bigwedge y. y < tg \implies \text{eval } ?\text{cond } [y, i, j, x] \downarrow \neq 0$  by simp
qed
moreover have eval r-parallel [i, j, x] =
  eval (Cn 3 r-both [m, Id 3 0, Id 3 1, Id 3 2]) [i, j, x]
  using m by simp
ultimately have eval r-parallel [i, j, x] = eval r-both [tg, i, j, x]
  using  $\langle \text{recfn } 3 \ m \rangle$  by simp
with * have eval r-parallel [i, j, x]  $\downarrow = \text{prod-encode } (1, \text{the } (\text{eval } g [x]))$ 
  by simp
then show ?thesis by simp
qed
next
assume eval-fg: eval g [x]  $\downarrow \wedge \text{eval } f [x] \uparrow$ 
then have i0:  $\forall t. \text{eval } r\text{-result1 } [t, i, x] \downarrow = 0$ 
  using r-result1-diverg' assms by auto
from eval-fg obtain v where eval g [x]  $\downarrow = v$ 
  by auto
then obtain  $t_0$  where  $t_0$ :
   $\forall t \geq t_0. \text{eval } r\text{-result1 } [t, j, x] \downarrow = \text{Suc } v$ 
   $\forall t < t_0. \text{eval } r\text{-result1 } [t, j, x] \downarrow = 0$ 
  using r-result1-converg' assms by metis

```

then have *: $eval\ r\text{-}both\ [t_0, i, j, x] \Downarrow = prod\text{-}encode\ (1, the\ (eval\ g\ [x]))$
using $r\text{-}both(4)$ *assms* $i0$ **by** *simp*
have $eval\ m\ [i, j, x] \Downarrow = t_0$
unfolding $m\text{-}def$
proof (*rule eval-Mn-convergI*)
show $recfn\ (length\ [i, j, x])\ (Mn\ 3\ ?cond)$ **by** *simp*
have $eval\ (Cn\ 4\ r\text{-}pdec1\ [r\text{-}both])\ [t_0, i, j, x] \Downarrow = 1$
using * **by** *simp*
then show $eval\ ?cond\ [t_0, i, j, x] \Downarrow = 0$ **by** *simp*
have $eval\ r\text{-}both\ [t, i, j, x] \Downarrow = prod\text{-}encode\ (2, 0)$ **if** $t < t_0$ **for** t
using $t0(2)$ $r\text{-}both(2)$ *assms* *that* $i0$ **by** *simp*
then have $eval\ ?cond\ [t, i, j, x] \Downarrow = 1$ **if** $t < t_0$ **for** t
using *that* **by** *simp*
then show $\bigwedge y. y < t_0 \implies eval\ ?cond\ [y, i, j, x] \Downarrow \neq 0$ **by** *simp*
qed
moreover have $eval\ r\text{-}parallel\ [i, j, x] =$
 $eval\ (Cn\ 3\ r\text{-}both\ [m, Id\ 3\ 0, Id\ 3\ 1, Id\ 3\ 2])\ [i, j, x]$
using m **by** *simp*
ultimately have $eval\ r\text{-}parallel\ [i, j, x] = eval\ r\text{-}both\ [t_0, i, j, x]$
using $\langle recfn\ 3\ m \rangle$ **by** *simp*
with * **show** $eval\ r\text{-}parallel\ [i, j, x] \Downarrow = prod\text{-}encode\ (1, the\ (eval\ g\ [x]))$
by *simp*
next
assume $eval\text{-}fg: eval\ f\ [x] \Downarrow \wedge eval\ g\ [x] \Uparrow$
then have $j0: \forall t. eval\ r\text{-}result1\ [t, j, x] \Downarrow = 0$
using $r\text{-}result1\text{-}diverg'$ *assms* **by** *auto*
from $eval\text{-}fg$ **obtain** v **where** $eval\ f\ [x] \Downarrow = v$
by *auto*
then obtain t_0 **where** $t0:$
 $\forall t \geq t_0. eval\ r\text{-}result1\ [t, i, x] \Downarrow = Suc\ v$
 $\forall t < t_0. eval\ r\text{-}result1\ [t, i, x] \Downarrow = 0$
using $r\text{-}result1\text{-}converg'$ *assms* **by** *metis*
then have *: $eval\ r\text{-}both\ [t_0, i, j, x] \Downarrow = prod\text{-}encode\ (0, the\ (eval\ f\ [x]))$
using $r\text{-}both(3)$ *assms* **by** *blast*
have $eval\ m\ [i, j, x] \Downarrow = t_0$
unfolding $m\text{-}def$
proof (*rule eval-Mn-convergI*)
show $recfn\ (length\ [i, j, x])\ (Mn\ 3\ ?cond)$ **by** *simp*
have $eval\ (Cn\ 4\ r\text{-}pdec1\ [r\text{-}both])\ [t_0, i, j, x] \Downarrow = 0$
using * **by** *simp*
then show $eval\ ?cond\ [t_0, i, j, x] \Downarrow = 0$
by *simp*
have $eval\ r\text{-}both\ [t, i, j, x] \Downarrow = prod\text{-}encode\ (2, 0)$ **if** $t < t_0$ **for** t
using $t0(2)$ $r\text{-}both(2)$ *assms* *that* $j0$ **by** *simp*
then have $eval\ ?cond\ [t, i, j, x] \Downarrow = 1$ **if** $t < t_0$ **for** t
using *that* **by** *simp*
then show $\bigwedge y. y < t_0 \implies eval\ ?cond\ [y, i, j, x] \Downarrow \neq 0$ **by** *simp*
qed
moreover have $eval\ r\text{-}parallel\ [i, j, x] =$
 $eval\ (Cn\ 3\ r\text{-}both\ [m, Id\ 3\ 0, Id\ 3\ 1, Id\ 3\ 2])\ [i, j, x]$
using m **by** *simp*
ultimately have $eval\ r\text{-}parallel\ [i, j, x] = eval\ r\text{-}both\ [t_0, i, j, x]$
using $\langle recfn\ 3\ m \rangle$ **by** *simp*
with * **show** $eval\ r\text{-}parallel\ [i, j, x] \Downarrow = prod\text{-}encode\ (0, the\ (eval\ f\ [x]))$
by *simp*
}

qed

end

theory *Standard-Results*

imports *Universal*

begin

1.7 Kleene normal form and the number of μ -operations

Kleene's original normal form theorem [11] states that every partial recursive f can be expressed as $f(x) = u(\mu y[t(i, x, y) = 0])$ for some i , where u and t are specially crafted primitive recursive functions tied to Kleene's definition of partial recursive functions. Rogers [12, p. 29f.] relaxes the theorem by allowing u and t to be any primitive recursive functions of arity one and three, respectively. Both versions require a separate t -predicate for every arity. We will show a unified version for all arities by treating x as an encoded list of arguments.

Our universal function

$r\text{-univ} \equiv$

$Cn\ 2\ r\text{-dec}\ [Cn\ 2\ r\text{-result}\ [Mn\ 2\ (Cn\ 3\ r\text{-not}\ [r\text{-result}]),\ Id\ 2\ 0,\ Id\ 2\ 1]]$

can represent all partial recursive functions (see theorem $r\text{-univ}$). Moreover $r\text{-result}$, $r\text{-dec}$, and $r\text{-not}$ are primitive recursive. As such $r\text{-univ}$ could almost serve as the right-hand side $u(\mu y[t(i, x, y) = 0])$. Its only flaw is that the outer function, the composition of $r\text{-dec}$ and $r\text{-result}$, is ternary rather than unary.

lemma $r\text{-univ-almost-kleene-nf}$:

$r\text{-univ} \simeq$

(let $u = Cn\ 3\ r\text{-dec}\ [r\text{-result}]$;

$t = Cn\ 3\ r\text{-not}\ [r\text{-result}]$

in $Cn\ 2\ u\ [Mn\ 2\ t,\ Id\ 2\ 0,\ Id\ 2\ 1]$)

unfolding $r\text{-univ-def}$ by (rule $exteqI$) simp-all

We can remedy the wrong arity with some encoding and projecting.

definition $r\text{-nf-t} :: \text{recf where}$

$r\text{-nf-t} \equiv Cn\ 3\ r\text{-and}$

$[Cn\ 3\ r\text{-eq}\ [Cn\ 3\ r\text{-pdec2}\ [Id\ 3\ 0],\ Cn\ 3\ r\text{-prod-encode}\ [Id\ 3\ 1,\ Id\ 3\ 2]],$

$Cn\ 3\ r\text{-not}$

$[Cn\ 3\ r\text{-result}$

$[Cn\ 3\ r\text{-pdec1}\ [Id\ 3\ 0],$

$Cn\ 3\ r\text{-pdec12}\ [Id\ 3\ 0],$

$Cn\ 3\ r\text{-pdec22}\ [Id\ 3\ 0]]]$

lemma $r\text{-nf-t-prim}$: prim-recfn 3 $r\text{-nf-t}$

unfolding $r\text{-nf-t-def}$ by simp

definition $r\text{-nf-u} :: \text{recf where}$

$r\text{-nf-u} \equiv Cn\ 1\ r\text{-dec}\ [Cn\ 1\ r\text{-result}\ [r\text{-pdec1},\ r\text{-pdec12},\ r\text{-pdec22}]]$

lemma $r\text{-nf-u-prim}$: prim-recfn 1 $r\text{-nf-u}$

unfolding $r\text{-nf-u-def}$ by simp

lemma $r\text{-nf-t-0}$:

assumes $eval\ r\text{-result}\ [pdec1\ y,\ pdec12\ y,\ pdec22\ y] \downarrow \neq 0$

and $pdec2\ y = prod\text{-}encode\ (i, x)$
shows $eval\ r\text{-}nf\text{-}t\ [y, i, x] \Downarrow = 0$
unfolding $r\text{-}nf\text{-}t\text{-}def$ **using** $assms$ **by** $auto$

lemma $r\text{-}nf\text{-}t\text{-}1$:

assumes $eval\ r\text{-}result\ [pdec1\ y, pdec12\ y, pdec22\ y] \Downarrow = 0 \vee pdec2\ y \neq prod\text{-}encode\ (i, x)$
shows $eval\ r\text{-}nf\text{-}t\ [y, i, x] \Downarrow = 1$
unfolding $r\text{-}nf\text{-}t\text{-}def$ **using** $assms\ r\text{-}result\text{-}total$ **by** $auto$

The next function is just as universal as $r\text{-}univ$, but satisfies the conditions of the Kleene normal form theorem because the outer function $r\text{-}nf\text{-}u$ is unary.

definition $r\text{-}normal\text{-}form \equiv Cn\ 2\ r\text{-}nf\text{-}u\ [Mn\ 2\ r\text{-}nf\text{-}t]$

lemma $r\text{-}normal\text{-}form\text{-}recfn$: $recfn\ 2\ r\text{-}normal\text{-}form$

unfolding $r\text{-}normal\text{-}form\text{-}def$ **using** $r\text{-}nf\text{-}u\text{-}prim\ r\text{-}nf\text{-}t\text{-}prim$ **by** $simp$

lemma $r\text{-}univ\text{-}exteq\text{-}r\text{-}normal\text{-}form$: $r\text{-}univ \simeq r\text{-}normal\text{-}form$

proof ($rule\ exteqI$)

show $arity\ r\text{-}univ = arity\ r\text{-}normal\text{-}form$

using $r\text{-}normal\text{-}form\text{-}recfn$ **by** $simp$

show $eval\ r\text{-}univ\ xs = eval\ r\text{-}normal\text{-}form\ xs$ **if** $length\ xs = arity\ r\text{-}univ$ **for** xs

proof –

have $length\ xs = 2$

using $that$ **by** $simp$

then obtain $i\ x$ **where** $ix: [i, x] = xs$

by ($smt\ Suc\text{-}length\text{-}conv\ length\text{-}0\text{-}conv\ numeral\text{-}2\text{-}eq\text{-}2$)

have $eval\ r\text{-}univ\ [i, x] = eval\ r\text{-}normal\text{-}form\ [i, x]$

proof ($cases\ \forall t. eval\ r\text{-}result\ [t, i, x] \Downarrow = 0$)

case $True$

then have $eval\ r\text{-}univ\ [i, x] \Uparrow$

unfolding $r\text{-}univ\text{-}def$ **by** $simp$

moreover have $eval\ r\text{-}normal\text{-}form\ [i, x] \Uparrow$

proof –

have $eval\ r\text{-}nf\text{-}t\ [y, i, x] \Downarrow = 1$ **for** y

using $True\ r\text{-}nf\text{-}t\text{-}1[of\ y\ i\ x]$ **by** $fastforce$

then show $?thesis$

unfolding $r\text{-}normal\text{-}form\text{-}def$ **using** $r\text{-}nf\text{-}u\text{-}prim\ r\text{-}nf\text{-}t\text{-}prim$ **by** $simp$

qed

ultimately show $?thesis$ **by** $simp$

next

case $False$

then have $\exists t. eval\ r\text{-}result\ [t, i, x] \Downarrow \neq 0$

by ($simp\ add: r\text{-}result\text{-}total$)

then obtain t **where** $eval\ r\text{-}result\ [t, i, x] \Downarrow \neq 0$

by $auto$

then have $eval\ r\text{-}nf\text{-}t\ [triple\text{-}encode\ t\ i\ x, i, x] \Downarrow = 0$

using $r\text{-}nf\text{-}t\text{-}0$ **by** $simp$

then obtain y **where** $y: eval\ (Mn\ 2\ r\text{-}nf\text{-}t)\ [i, x] \Downarrow = y$

using $r\text{-}nf\text{-}t\text{-}prim\ Mn\text{-}free\text{-}imp\text{-}total$ **by** $fastforce$

then have $eval\ r\text{-}nf\text{-}t\ [y, i, x] \Downarrow = 0$

using $r\text{-}nf\text{-}t\text{-}prim\ Mn\text{-}free\text{-}imp\text{-}total\ eval\text{-}Mn\text{-}convergE(2)[of\ 2\ r\text{-}nf\text{-}t\ [i, x]\ y]$

by $simp$

then have $r\text{-}result$: $eval\ r\text{-}result\ [pdec1\ y, pdec12\ y, pdec22\ y] \Downarrow \neq 0$

and $pdec2$: $pdec2\ y = prod\text{-}encode\ (i, x)$

using $r\text{-}nf\text{-}t\text{-}0[of\ y\ i\ x]\ r\text{-}nf\text{-}t\text{-}1[of\ y\ i\ x]\ r\text{-}result\text{-}total$ **by** $auto$

then have $eval\ r\text{-}result\ [pdec1\ y, i, x] \Downarrow \neq 0$

by simp
then obtain v where v :
 eval r-univ [pdec12 y , pdec22 y] $\downarrow = v$
 eval r-result [pdec1 y , pdec12 y , pdec22 y] $\downarrow = \text{Suc } v$
using *r-result r-result-bivalent'*[of pdec12 y pdec22 y - pdec1 y]
 r-result-diverg'[of pdec12 y pdec22 y pdec1 y]
by auto

have *eval r-normal-form* [i , x] = *eval r-nf-u* [y]
 unfolding *r-normal-form-def* **using** y *r-nf-t-prim r-nf-u-prim* **by simp**
also have ... = *eval r-dec* [the (*eval* (Cn 1 *r-result* [r-pdec1, r-pdec12, r-pdec22]) [y])])
 unfolding *r-nf-u-def* **using** *r-result* **by simp**
also have ... = *eval r-dec* [Suc v]
 using v **by simp**
also have ... $\downarrow = v$
 by simp
finally have *eval r-normal-form* [i , x] $\downarrow = v$.
moreover have *eval r-univ* [i , x] $\downarrow = v$
 using $v(1)$ pdec2 **by simp**
ultimately show ?thesis **by simp**
qed
with ix show ?thesis **by simp**
qed
qed

theorem normal-form:
 assumes *recfn* n f
 obtains i **where** $\forall x$. *e-length* $x = n \longrightarrow \text{eval } r\text{-normal-form } [i, x] = \text{eval } f (\text{list-decode } x)$
proof –
 have *eval r-normal-form* [encode f , x] = *eval f* (list-decode x) **if** *e-length* $x = n$ **for** x
 using *r-univ-exteq-r-normal-form* *assms* that *exteq-def r-univ'* **by auto**
 then show ?thesis **using that** **by auto**
qed

As a consequence of the normal form theorem every partial recursive function can be represented with exactly one application of the μ -operator.

fun *count-Mn* :: *recf* \Rightarrow *nat* **where**
 count-Mn $Z = 0$
| *count-Mn* $S = 0$
| *count-Mn* (Id m n) = 0
| *count-Mn* (Cn n f gs) = *count-Mn* f + *sum-list* (map *count-Mn* gs)
| *count-Mn* (Pr n f g) = *count-Mn* f + *count-Mn* g
| *count-Mn* (Mn n f) = Suc (*count-Mn* f)

lemma *count-Mn-zero-iff-prim*: *count-Mn* $f = 0 \longleftrightarrow$ *Mn-free* f
 by (*induction* f) *auto*

The normal form has only one μ -recursion.

lemma *count-Mn-normal-form*: *count-Mn r-normal-form* = 1
 unfolding *r-normal-form-def r-nf-u-def r-nf-t-def* **using** *count-Mn-zero-iff-prim* **by simp**

lemma *one-Mn-suffices*:
 assumes *recfn* n f
 shows $\exists g$. *count-Mn* $g = 1 \wedge g \simeq f$
proof –

have $n > 0$
using *assms wellf-arity-nonzero* **by** *auto*
obtain i **where** i :
 $\forall x. e\text{-length } x = n \longrightarrow eval\ r\text{-normal-form } [i, x] = eval\ f\ (list\text{-decode } x)$
using *normal-form[OF assms(1)]* **by** *auto*
define g **where** $g \equiv Cn\ n\ r\text{-normal-form } [r\text{-constn } (n - 1)\ i, r\text{-list-encode } (n - 1)]$
then have *recfn n g*
using *r-normal-form-recfn <n > 0>* **by** *simp*
then have $g \simeq f$
using *g-def r-list-encode i assms* **by** (*intro exteqI*) *simp-all*
moreover have *count-Mn g = 1*
unfolding *g-def* **using** *count-Mn-normal-form count-Mn-zero-iff-prim* **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

The previous lemma could have been obtained without *r-normal-form* directly from *r-univ*.

1.8 The s - m - n theorem

For all $m, n > 0$ there is an $(m + 1)$ -ary primitive recursive function s_n^m with

$$\varphi_p^{(m+n)}(c_1, \dots, c_m, x_1, \dots, x_n) = \varphi_{s_n^m(p, c_1, \dots, c_m)}^{(n)}(x_1, \dots, x_n)$$

for all $p, c_1, \dots, c_m, x_1, \dots, x_n$. Here, $\varphi^{(n)}$ is a function universal for n -ary partial recursive functions, which we will represent by *r-universal n*

The s_n^m functions compute codes of functions. We start simple: computing codes of the unary constant functions.

fun *code-const1* :: *nat* \Rightarrow *nat* **where**
code-const1 0 = 0
| *code-const1 (Suc c) = quad-encode 3 1 1 (singleton-encode (code-const1 c))*

lemma *code-const1*: *code-const1 c = encode (r-const c)*
by (*induction c*) *simp-all*

definition *r-code-const1-aux* \equiv
Cn 3 r-prod-encode
[*r-constn 2 3*,
Cn 3 r-prod-encode
[*r-constn 2 1*,
Cn 3 r-prod-encode
[*r-constn 2 1*, *Cn 3 r-singleton-encode [Id 3 1]]]]]*

lemma *r-code-const1-aux-prim*: *prim-recfn 3 r-code-const1-aux*
by (*simp-all add: r-code-const1-aux-def*)

lemma *r-code-const1-aux*:
eval r-code-const1-aux [i, r, c] \Downarrow= quad-encode 3 1 1 (singleton-encode r)
by (*simp add: r-code-const1-aux-def*)

definition *r-code-const1* \equiv *r-shrink (Pr 1 Z r-code-const1-aux)*

lemma *r-code-const1-prim*: *prim-recfn 1 r-code-const1*

by (simp-all add: r-code-const1-def r-code-const1-aux-prim)

lemma r-code-const1: eval r-code-const1 [c] \downarrow = code-const1 c

proof –

let ?h = Pr 1 Z r-code-const1-aux

have eval ?h [c, x] \downarrow = code-const1 c for x

using r-code-const1-aux r-code-const1-def

by (induction c) (simp-all add: r-code-const1-aux-prim)

then show ?thesis by (simp add: r-code-const1-def r-code-const1-aux-prim)

qed

Functions that compute codes of higher-arity constant functions:

definition code-constn :: nat \Rightarrow nat \Rightarrow nat **where**

code-constn n c \equiv

if n = 1 then code-const1 c

else quad-encode 3 n (code-const1 c) (singleton-encode (triple-encode 2 n 0))

lemma code-constn: code-constn (Suc n) c = encode (r-constn n c)

unfolding code-constn-def **using** code-const1 r-constn-def

by (cases n = 0) simp-all

definition r-code-constn :: nat \Rightarrow recf **where**

r-code-constn n \equiv

if n = 1 then r-code-const1

else

Cn 1 r-prod-encode

[r-const 3,

Cn 1 r-prod-encode

[r-const n,

Cn 1 r-prod-encode

[r-code-const1,

Cn 1 r-singleton-encode

[Cn 1 r-prod-encode

[r-const 2, Cn 1 r-prod-encode [r-const n, Z]]]]]]

lemma r-code-constn-prim: prim-recfn 1 (r-code-constn n)

by (simp-all add: r-code-constn-def r-code-const1-prim)

lemma r-code-constn: eval (r-code-constn n) [c] \downarrow = code-constn n c

by (auto simp add: r-code-constn-def r-code-const1 code-constn-def r-code-const1-prim)

Computing codes of m -ary projections:

definition code-id :: nat \Rightarrow nat \Rightarrow nat **where**

code-id m n \equiv triple-encode 2 m n

lemma code-id: encode (Id m n) = code-id m n

unfolding code-id-def **by** simp

The functions s_n^m are represented by the following function. The value m corresponds to the length of cs .

definition smn :: nat \Rightarrow nat \Rightarrow nat list \Rightarrow nat **where**

smn n p cs \equiv quad-encode

3

n

(encode (r-universal (n + length cs)))

(list-encode (code-constn n p # map (code-constn n) cs @ map (code-id n) [0..<n]))

lemma smn:

assumes $n > 0$

shows $smn\ n\ p\ cs = encode$

($Cn\ n$

(r -universal ($n + length\ cs$))

(r -constn ($n - 1$) p # map (r -constn ($n - 1$)) cs @ (map ($Id\ n$) [0..<n])))

proof –

let $?p = r$ -constn ($n - 1$) p

let $?gs1 = map\ (r$ -constn ($n - 1$)) cs

let $?gs2 = map\ (Id\ n)\ [0..<n]$

let $?gs = ?p\ \#\ ?gs1\ \@\ ?gs2$

have map encode $?gs1 = map\ (code$ -constn $n)\ cs$

by (intro nth-equalityI; auto; metis code-constn assms Suc-pred)

moreover **have** map encode $?gs2 = map\ (code$ -id $n)\ [0..<n]$

by (rule nth-equalityI) (auto simp add: code-id-def)

moreover **have** encode $?p = code$ -constn $n\ p$

using assms code-constn[of $n - 1\ p$] **by** simp

ultimately **have** map encode $?gs =$

code-constn $n\ p\ \#\ map\ (code$ -constn $n)\ cs\ \@\ map\ (code$ -id $n)\ [0..<n]$

by simp

then **show** $?thesis$

unfolding smn-def **using** assms encode.simps(4) **by** presburger

qed

The next function is to help us define *recfs* corresponding to the s_n^m functions. It maps $m+1$ arguments p, c_1, \dots, c_m to an encoded list of length $m+n+1$. The list comprises the $m+1$ codes of the n -ary constants p, c_1, \dots, c_m and the n codes for all n -ary projections.

definition r -smn-aux :: $nat \Rightarrow nat \Rightarrow recf$ **where**

r -smn-aux $n\ m \equiv$

$Cn\ (Suc\ m)$

(r -list-encode ($m + n$))

(map ($\lambda i.$ $Cn\ (Suc\ m)\ (r$ -code-constn $n)\ [Id\ (Suc\ m)\ i])\ [0..<Suc\ m]\ \@\$

map ($\lambda i.$ r -constn $m\ (code$ -id $n\ i))\ [0..<n]$)

lemma r -smn-aux-prim: $n > 0 \implies prim$ -recfn ($Suc\ m$) (r -smn-aux $n\ m$)

by (auto simp add: r -smn-aux-def r -code-constn-prim)

lemma r -smn-aux:

assumes $n > 0$ **and** length $cs = m$

shows eval (r -smn-aux $n\ m$) ($p\ \#\ cs$) $\downarrow =$

list-encode (map (code-constn n) ($p\ \#\ cs$) @ map (code-id n) [0..<n])

proof –

let $?xs = map\ (\lambda i.$ $Cn\ (Suc\ m)\ (r$ -code-constn $n)\ [Id\ (Suc\ m)\ i])\ [0..<Suc\ m]$

let $?ys = map\ (\lambda i.$ r -constn $m\ (code$ -id $n\ i))\ [0..<n]$

have len- xs : length $?xs = Suc\ m$ **by** simp

have map- xs : map ($\lambda g.$ eval $g\ (p\ \#\ cs)$) $?xs = map\ Some\ (map\ (code$ -constn $n)\ (p\ \#\ cs))$

proof (intro nth-equalityI)

show len: length (map ($\lambda g.$ eval $g\ (p\ \#\ cs)$) $?xs$) =

length (map Some (map (code-constn n) ($p\ \#\ cs$)))

by (simp add: assms(2))

have map ($\lambda g.$ eval $g\ (p\ \#\ cs)$) $?xs\ !\ i = map\ Some\ (map\ (code$ -constn $n)\ (p\ \#\ cs))\ !\ i$

if $i < Suc\ m$ **for** i

proof –

- have** $\text{map } (\lambda g. \text{eval } g (p \# cs)) ?xs ! i = (\lambda g. \text{eval } g (p \# cs)) (?xs ! i)$
- using** *len-xs that* **by** (*metis nth-map*)
- also have** $\dots = \text{eval } (Cn (Suc m) (r\text{-code-constn } n) [Id (Suc m) i]) (p \# cs)$
- using** *that len-xs*
- by** (*metis (no-types, lifting) add.left-neutral length-map nth-map nth-upt*)
- also have** $\dots = \text{eval } (r\text{-code-constn } n) [\text{the } (\text{eval } (Id (Suc m) i) (p \# cs))]$
- using** *r-code-constn-prim assms(2) that* **by** *simp*
- also have** $\dots = \text{eval } (r\text{-code-constn } n) [(p \# cs) ! i]$
- using** *len that* **by** *simp*
- finally have** $\text{map } (\lambda g. \text{eval } g (p \# cs)) ?xs ! i \downarrow = \text{code-constn } n ((p \# cs) ! i)$
- using** *r-code-constn* **by** *simp*
- then show** *?thesis*
- using** *len-xs len that* **by** (*metis length-map nth-map*)

qed

- moreover have** $\text{length } (\text{map } (\lambda g. \text{eval } g (p \# cs)) ?xs) = Suc m$ **by** *simp*
- ultimately show** $\bigwedge i. i < \text{length } (\text{map } (\lambda g. \text{eval } g (p \# cs)) ?xs) \implies$
- $\text{map } (\lambda g. \text{eval } g (p \# cs)) ?xs ! i =$
- $\text{map } \text{Some } (\text{map } (\text{code-constn } n) (p \# cs)) ! i$
- by** *simp*

qed

- moreover have** $\text{map } (\lambda g. \text{eval } g (p \# cs)) ?ys = \text{map } \text{Some } (\text{map } (\text{code-id } n) [0..<n])$
- using** *assms(2) by (intro nth-equalityI; auto)*
- ultimately have** $\text{map } (\lambda g. \text{eval } g (p \# cs)) (?xs @ ?ys) =$
- $\text{map } \text{Some } (\text{map } (\text{code-constn } n) (p \# cs) @ \text{map } (\text{code-id } n) [0..<n])$
- by** (*metis map-append*)
- moreover have** $\text{map } (\lambda x. \text{the } (\text{eval } x (p \# cs))) (?xs @ ?ys) =$
- $\text{map } \text{the } (\text{map } (\lambda x. \text{eval } x (p \# cs)) (?xs @ ?ys))$
- by** *simp*
- ultimately have** $*$: $\text{map } (\lambda g. \text{the } (\text{eval } g (p \# cs))) (?xs @ ?ys) =$
- $(\text{map } (\text{code-constn } n) (p \# cs) @ \text{map } (\text{code-id } n) [0..<n])$
- by** *simp*

have $\forall i < \text{length } ?xs. \text{eval } (?xs ! i) (p \# cs) = \text{map } (\lambda g. \text{eval } g (p \# cs)) ?xs ! i$

by (*metis nth-map*)

then have

- $\forall i < \text{length } ?xs. \text{eval } (?xs ! i) (p \# cs) = \text{map } \text{Some } (\text{map } (\text{code-constn } n) (p \# cs)) ! i$
- using** *map-xs* **by** *simp*

then have $\forall i < \text{length } ?xs. \text{eval } (?xs ! i) (p \# cs) \downarrow$

using *assms map-xs* **by** (*metis length-map nth-map option.simps(3)*)

then have *xs-converg*: $\forall z \in \text{set } ?xs. \text{eval } z (p \# cs) \downarrow$

by (*metis in-set-conv-nth*)

have $\forall i < \text{length } ?ys. \text{eval } (?ys ! i) (p \# cs) = \text{map } (\lambda x. \text{eval } x (p \# cs)) ?ys ! i$

by *simp*

then have

- $\forall i < \text{length } ?ys. \text{eval } (?ys ! i) (p \# cs) = \text{map } \text{Some } (\text{map } (\text{code-id } n) [0..<n]) ! i$
- using** *assms(2) by simp*

then have $\forall i < \text{length } ?ys. \text{eval } (?ys ! i) (p \# cs) \downarrow$

by *simp*

then have $\forall z \in \text{set } (?xs @ ?ys). \text{eval } z (p \# cs) \downarrow$

using *xs-converg* **by** *auto*

moreover have *recfn* $(\text{length } (p \# cs)) (Cn (Suc m) (r\text{-list-encode } (m + n)) (?xs @ ?ys))$

using *assms r-code-constn-prim* **by** *auto*

ultimately have $\text{eval } (r\text{-smn-aux } n m) (p \# cs) =$

$\text{eval } (r\text{-list-encode } (m + n)) (\text{map } (\lambda g. \text{the } (\text{eval } g (p \# cs))) (?xs @ ?ys))$

unfolding *r-smn-aux-def* **using** *assms* **by** *simp*
then have *eval (r-smn-aux n m) (p # cs) =*
eval (r-list-encode (m + n)) (map (code-constn n) (p # cs) @ map (code-id n) [0..<n])
using * **by** *metis*
moreover have *length (?xs @ ?ys) = Suc (m + n)* **by** *simp*
ultimately show *?thesis*
using *r-list-encode * assms(1)* **by** (*metis (no-types, lifting) length-map*)
qed

For all $m, n > 0$, the *recf* corresponding to s_n^m is given by the next function.

definition *r-smn* :: *nat* \Rightarrow *nat* \Rightarrow *recf* **where**

r-smn n m \equiv
Cn (Suc m) r-prod-encode
[r-constn m 3,
Cn (Suc m) r-prod-encode
[r-constn m n,
Cn (Suc m) r-prod-encode
[r-constn m (encode (r-universal (n + m))), r-smn-aux n m]]]

lemma *r-smn-prim [simp]: n > 0 \implies prim-recfn (Suc m) (r-smn n m)*
by (*simp-all add: r-smn-def r-smn-aux-prim*)

lemma *r-smn:*

assumes *n > 0 and length cs = m*
shows *eval (r-smn n m) (p # cs) \downarrow = smn n p cs*
using *assms r-smn-def r-smn-aux smn-def r-smn-aux-prim* **by** *simp*

lemma *map-eval-Some-the:*

assumes *map ($\lambda g. eval g xs$) gs = map Some ys*
shows *map ($\lambda g. the (eval g xs)$) gs = ys*
using *assms*
by (*metis (no-types, lifting) length-map nth-equalityI nth-map option.sel*)

The essential part of the *s-m-n* theorem: For all $m, n > 0$ the function s_n^m satisfies

$$\varphi_p^{(m+n)}(c_1, \dots, c_m, x_1, \dots, x_n) = \varphi_{s_n^m(p, c_1, \dots, c_m)}^{(n)}(x_1, \dots, x_n)$$

for all p, c_i, x_j .

lemma *smn-lemma:*

assumes *n > 0 and len-cs: length cs = m and len-xs: length xs = n*
shows *eval (r-universal (m + n)) (p # cs @ xs) =*
eval (r-universal n) ((the (eval (r-smn n m) (p # cs))) # xs)

proof –

let *?s = r-smn n m*
let *?f = Cn n*
(r-universal (n + length cs))
(r-constn (n - 1) p # map (r-constn (n - 1)) cs @ (map (Id n) [0..<n]))
have *eval ?s (p # cs) \downarrow = smn n p cs*
using *assms r-smn* **by** *simp*
then have *eval-s: eval ?s (p # cs) \downarrow = encode ?f*
by (*simp add: assms(1) smn*)

have *recfn n ?f*

using *len-cs assms* **by** *auto*

then have *: *eval (r-universal n) ((encode ?f) # xs) = eval ?f xs*

```

using r-universal[of ?f n, OF - len-xs] by simp

let ?gs = r-constn (n - 1) p # map (r-constn (n - 1)) cs @ map (Id n) [0..<n]
have  $\forall g \in \text{set } ?gs. \text{eval } g \text{ xs} \downarrow$ 
  using len-cs len-xs assms by auto
then have eval ?f xs =
  eval (r-universal (n + length cs)) (map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs)
  using len-cs len-xs assms <recfn n ?f> by simp
then have eval ?f xs = eval (r-universal (m + n)) (map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs)
  by (simp add: len-cs add.commute)
then have eval (r-universal n) ((the (eval ?s (p # cs))) # xs) =
  eval (r-universal (m + n)) (map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs)
  using eval-s * by simp
moreover have map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs = p # cs @ xs
proof (intro nth-equalityI)
  show length (map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs) = length (p # cs @ xs)
    by (simp add: len-xs)
  have len: length (map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs) = Suc (m + n)
    by (simp add: len-cs)
  moreover have map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs ! i = (p # cs @ xs) ! i
    if  $i < \text{Suc } (m + n)$  for  $i$ 
  proof -
    from that consider i = 0 | i > 0 ∧ i < Suc m | Suc m ≤ i ∧ i < Suc (m + n)
      using not-le-imp-less by auto
    then show ?thesis
    proof (cases)
      case 1
        then show ?thesis using assms(1) len-xs by simp
      next
        case 2
          then have ?gs ! i = (map (r-constn (n - 1)) cs) ! (i - 1)
            using len-cs
            by (metis One-nat-def Suc-less-eq Suc-pred length-map less-numeral-extra(3) nth-Cons' nth-append)
          then have map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs ! i =
            ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ((map (r-constn (n - 1)) cs) ! (i - 1))
            using len by (metis length-map nth-map that)
          also have ... = the (eval ((r-constn (n - 1)) (cs ! (i - 1)))) xs
            using 2 len-cs by auto
          also have ... = cs ! (i - 1)
            using r-constn len-xs assms(1) by simp
          also have ... = (p # cs @ xs) ! i
            using 2 len-cs
            by (metis diff-Suc-1 less-Suc-eq-0-disj less-numeral-extra(3) nth-Cons' nth-append)
          finally show ?thesis .
        next
          case 3
            then have ?gs ! i = (map (Id n) [0..<n]) ! (i - Suc m)
              using len-cs
              by (simp; metis (no-types, lifting) One-nat-def Suc-less-eq add-leE plus-1-eq-Suc diff-diff-left length-map not-le nth-append ordered-cancel-comm-monoid-diff-class.add-diff-inverse)
            then have map ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ?gs ! i =
              ( $\lambda g. \text{the (eval } g \text{ xs)}$ ) ((map (Id n) [0..<n]) ! (i - Suc m))
              using len by (metis length-map nth-map that)
            also have ... = the (eval ((Id n (i - Suc m)))) xs

```

```

    using 3 len-cs by auto
  also have ... = xs ! (i - Suc m)
    using len-xs 3 by auto
  also have ... = (p # cs @ xs) ! i
    using len-cs len-xs 3
    by (metis diff-Suc-1 diff-diff-left less-Suc-eq-0-disj not-le nth-Cons'
        nth-append plus-1-eq-Suc)
  finally show ?thesis .
qed
qed
ultimately show map (λg. the (eval g xs)) ?gs ! i = (p # cs @ xs) ! i
  if i < length (map (λg. the (eval g xs)) ?gs) for i
  using that by simp
qed
ultimately show ?thesis by simp
qed

```

theorem smn-theorem:

```

  assumes n > 0
  shows ∃ s. prim-recfn (Suc m) s ∧
    (∀ p cs xs. length cs = m ∧ length xs = n →
      eval (r-universal (m + n)) (p # cs @ xs) =
      eval (r-universal n) ((the (eval s (p # cs))) # xs))
  using smn-lemma exI[of - r-smn n m] assms by simp

```

For every numbering, that is, binary partial recursive function, ψ there is a total recursive function c that translates ψ -indices into φ -indices.

lemma numbering-translation:

```

  assumes recfn 2 psi
  obtains c where
    recfn 1 c
    total c
    ∀ i x. eval psi [i, x] = eval r-phi [the (eval c [i]), x]

```

proof –

```

  let ?p = encode psi
  define c where c = Cn 1 (r-smn 1 1) [r-const ?p, Id 1 0]
  then have prim-recfn 1 c by simp
  moreover from this have total c
    by auto
  moreover have eval r-phi [the (eval c [i]), x] = eval psi [i, x] for i x

```

proof –

```

  have eval c [i] = eval (r-smn 1 1) [?p, i]
    using c-def by simp
  then have eval (r-universal 1) [the (eval c [i]), x] =
    eval (r-universal 1) [the (eval (r-smn 1 1) [?p, i]), x]
    by simp
  also have ... = eval (r-universal (1 + 1)) (?p # [i] @ [x])
    using smn-lemma[of 1 [i] 1 [x] ?p] by simp
  also have ... = eval (r-universal 2) [?p, i, x]
    by (metis append-eq-Cons-conv nat-1-add-1)
  also have ... = eval psi [i, x]
    using r-universal[OF assms, of [i, x]] by simp
  finally have eval (r-universal 1) [the (eval c [i]), x] = eval psi [i, x] .
  then show ?thesis using r-phi-def by simp

```

qed

ultimately show ?thesis using that by auto

qed

1.9 Fixed-point theorems

Fixed-point theorems (also known as recursion theorems) come in many shapes. We prove the minimum we need for Chapter 2.

1.9.1 Rogers's fixed-point theorem

In this section we prove a theorem that Rogers [12] credits to Kleene, but admits that it is a special case and not the original formulation. We follow Wikipedia [17] and call it the Rogers's fixed-point theorem.

lemma *s11-inj*: $\text{inj } (\lambda x. \text{smn } 1 \text{ } p \text{ } [x])$

proof

fix $x_1 \ x_2 :: \text{nat}$

assume $\text{smn } 1 \text{ } p \text{ } [x_1] = \text{smn } 1 \text{ } p \text{ } [x_2]$

then have $\text{list-encode } [\text{code-constn } 1 \text{ } p, \text{code-constn } 1 \text{ } x_1, \text{code-id } 1 \text{ } 0] =$
 $\text{list-encode } [\text{code-constn } 1 \text{ } p, \text{code-constn } 1 \text{ } x_2, \text{code-id } 1 \text{ } 0]$

using *smn-def* **by** (*simp add: prod-encode-eq*)

then have $[\text{code-constn } 1 \text{ } p, \text{code-constn } 1 \text{ } x_1, \text{code-id } 1 \text{ } 0] =$
 $[\text{code-constn } 1 \text{ } p, \text{code-constn } 1 \text{ } x_2, \text{code-id } 1 \text{ } 0]$

using *list-decode-encode* **by** *metis*

then have $\text{code-constn } 1 \text{ } x_1 = \text{code-constn } 1 \text{ } x_2$ **by** *simp*

then show $x_1 = x_2$

using *code-const1 code-constn code-constn-def encode-injective r-constn*
by (*metis One-nat-def length-Cons list.size(3) option.simps(1)*)

qed

definition *r-univuniv* $\equiv \text{Cn } 2 \text{ } r\text{-phi } [\text{Id } 2 \text{ } 0, \text{Id } 2 \text{ } 0], \text{Id } 2 \text{ } 1]$

lemma *r-univuniv-recfn*: $\text{recfn } 2 \text{ } r\text{-univuniv}$

by (*simp add: r-univuniv-def*)

lemma *r-univuniv-converg*:

assumes $\text{eval } r\text{-phi } [x, x] \downarrow$

shows $\text{eval } r\text{-univuniv } [x, y] = \text{eval } r\text{-phi } [\text{the } (\text{eval } r\text{-phi } [x, x]), y]$

unfolding *r-univuniv-def* **using** *assms r-univuniv-recfn r-phi-recfn* **by** *simp*

Strictly speaking this is a generalization of Rogers's theorem in that it shows the existence of infinitely many fixed-points. In conventional terms it says that for every total recursive f and $k \in \mathbb{N}$ there is an $n \geq k$ with $\varphi_n = \varphi_{f(n)}$.

theorem *rogers-fixed-point-theorem*:

fixes $k :: \text{nat}$

assumes $\text{recfn } 1 \text{ } f$ **and** *total f*

shows $\exists n \geq k. \forall x. \text{eval } r\text{-phi } [n, x] = \text{eval } r\text{-phi } [\text{the } (\text{eval } f \text{ } [n]), x]$

proof –

let $?p = \text{encode } r\text{-univuniv}$

define h **where** $h = \text{Cn } 1 \text{ } (r\text{-smn } 1 \text{ } 1) \text{ } [r\text{-const } ?p, \text{Id } 1 \text{ } 0]$

then have *prim-recfn 1 h*

by *simp*

then have *total h*

by *blast*

have $\text{eval } h \text{ } [x] = \text{eval } (\text{Cn } 1 \text{ } (r\text{-smn } 1 \text{ } 1) \text{ } [r\text{-const } ?p, \text{Id } 1 \text{ } 0]) \text{ } [x]$ **for** x

unfolding h -def **by** *simp*
then have h : $the (eval\ h\ [x]) = smn\ 1\ ?p\ [x]$ **for** x
by (*simp add: r-smn*)

have $eval\ r$ -phi [$the (eval\ h\ [x]), y$] = $eval\ r$ -univuniv [x, y] **for** $x\ y$
proof –
have $eval\ r$ -phi [$the (eval\ h\ [x]), y$] = $eval\ r$ -phi [$smn\ 1\ ?p\ [x], y$]
using h **by** *simp*
also have ... = $eval\ r$ -phi [$the (eval\ (r-smn\ 1\ 1)\ [?p, x]), y$]
by (*simp add: r-smn*)
also have ... = $eval\ (r$ -universal 2) [$?p, x, y$]
using r -phi-def smn -lemma[*of 1 [x] 1 [y] ?p*]
by (*metis Cons-eq-append-conv One-nat-def Suc-1 length-Cons less-numeral-extra(1) list.size(3) plus-1-eq-Suc*)
finally show $eval\ r$ -phi [$the (eval\ h\ [x]), y$] = $eval\ r$ -univuniv [x, y]
using r -universal r -univuniv-recfn **by** *simp*
qed

then have *: $eval\ r$ -phi [$the (eval\ h\ [x]), y$] = $eval\ r$ -phi [$the (eval\ r$ -phi [$x, x]), y$]
if $eval\ r$ -phi [x, x] \downarrow **for** $x\ y$
using r -univuniv-converg **that** **by** *simp*

let $?fh = Cn\ 1\ f\ [h]$
have $recfn\ 1\ ?fh$
using $\langle prim$ -recfn 1 $h \rangle$ *assms* **by** *simp*
then have $infinite\ \{r. recfn\ 1\ r \wedge r \simeq ?fh\}$
using *exteq-infinite[of ?fh 1]* **by** *simp*
then have $infinite\ (encode\ \{r. recfn\ 1\ r \wedge r \simeq ?fh\})$ (**is** $infinite\ ?E$)
using *encode-injective* **by** (*meson finite-imageD inj-onI*)
then have $infinite\ ((\lambda x. smn\ 1\ ?p\ [x])\ \{?E\})$
using *s11-inj[of ?p]* **by** (*simp add: finite-image-iff inj-on-subset*)
moreover have $(\lambda x. smn\ 1\ ?p\ [x])\ \{?E\} = \{smn\ 1\ ?p\ [encode\ r] \mid r. recfn\ 1\ r \wedge r \simeq ?fh\}$
by *auto*
ultimately have $infinite\ \{smn\ 1\ ?p\ [encode\ r] \mid r. recfn\ 1\ r \wedge r \simeq ?fh\}$
by *simp*
then obtain n **where** $n \geq k\ n \in \{smn\ 1\ ?p\ [encode\ r] \mid r. recfn\ 1\ r \wedge r \simeq ?fh\}$
by (*meson finite-nat-set-iff-bounded-le le-cases*)
then obtain r **where** $r: recfn\ 1\ r\ n = smn\ 1\ ?p\ [encode\ r]\ recfn\ 1\ r \wedge r \simeq ?fh$
by *auto*
then have $eval$ - r : $eval\ r\ [encode\ r] = eval\ ?fh\ [encode\ r]$
by (*simp add: exteq-def*)
then have $eval$ - r' : $eval\ r\ [encode\ r] = eval\ f\ [the\ (eval\ h\ [encode\ r])]$
using *assms* $\langle total\ h \rangle$ $\langle prim$ -recfn 1 $h \rangle$ **by** *simp*
then have $eval\ r\ [encode\ r] \downarrow$
using $\langle prim$ -recfn 1 $h \rangle$ *assms*(1,2) **by** *simp*
then have $eval\ r$ -phi [$encode\ r, encode\ r$] \downarrow
by (*simp add: recfn 1 r r-phi*)
then have $eval\ r$ -phi [$the (eval\ h\ [encode\ r]), y$] =
 $eval\ r$ -phi [($the (eval\ r$ -phi [$encode\ r, encode\ r$)]), y]
for y
using * **by** *simp*
then have $eval\ r$ -phi [$the (eval\ h\ [encode\ r]), y$] =
 $eval\ r$ -phi [($the (eval\ r\ [encode\ r])$), y]
for y
by (*simp add: recfn 1 r r-phi*)
moreover have $n = the\ (eval\ h\ [encode\ r])$ **by** (*simp add: h r(2)*)
ultimately have $eval\ r$ -phi [n, y] = $eval\ r$ -phi [$the (eval\ r\ [encode\ r]), y$] **for** y

by *simp*
then have $eval\ r\text{-}phi\ [n, y] = eval\ r\text{-}phi\ [the\ (eval\ ?fh\ [encode\ r]), y]$ **for** y
 using r by (*simp add: eval-r*)
moreover have $eval\ ?fh\ [encode\ r] = eval\ f\ [n]$
 using $eval\text{-}r\ eval\text{-}r'\ \langle n = the\ (eval\ h\ [encode\ r]) \rangle$ **by** *auto*
ultimately have $eval\ r\text{-}phi\ [n, y] = eval\ r\text{-}phi\ [the\ (eval\ f\ [n]), y]$ **for** y
 by *simp*
with $\langle n \geq k \rangle$ **show** *?thesis* **by** *auto*
qed

1.9.2 Kleene's fixed-point theorem

The next theorem is what Rogers [12, p. 214] calls Kleene's version of what we call Rogers's fixed-point theorem. More precisely this would be Kleene's *second* fixed-point theorem, but since we do not cover the first one, we leave out the number.

theorem *kleene-fixed-point-theorem:*

fixes $k :: nat$
assumes *recfn 2 psi*
shows $\exists n \geq k. \forall x. eval\ r\text{-}phi\ [n, x] = eval\ psi\ [n, x]$
proof –
from *numbering-translation[OF assms]* **obtain** c **where** c :
 $recfn\ 1\ c$
 $total\ c$
 $\forall i\ x. eval\ psi\ [i, x] = eval\ r\text{-}phi\ [the\ (eval\ c\ [i]), x]$
by *auto*
then obtain n **where** $n \geq k$ **and** $\forall x. eval\ r\text{-}phi\ [n, x] = eval\ r\text{-}phi\ [the\ (eval\ c\ [n]), x]$
using *rogers-fixed-point-theorem* **by** *blast*
with $c(3)$ **have** $\forall x. eval\ r\text{-}phi\ [n, x] = eval\ psi\ [n, x]$
by *simp*
with $\langle n \geq k \rangle$ **show** *?thesis* **by** *auto*
qed

Kleene's fixed-point theorem can be generalized to arbitrary arities. But we need to generalize it only to binary functions in order to show Smullyan's double fixed-point theorem in Section 1.9.3.

definition *r-univuniv2* \equiv

$Cn\ 3\ r\text{-}phi\ [Cn\ 3\ (r\text{-}universal\ 2)\ [Id\ 3\ 0, Id\ 3\ 0, Id\ 3\ 1], Id\ 3\ 2]$

lemma *r-univuniv2-recfn: recfn 3 r-univuniv2*

by (*simp add: r-univuniv2-def*)

lemma *r-univuniv2-converg:*

assumes $eval\ (r\text{-}universal\ 2)\ [u, u, x] \downarrow$
shows $eval\ r\text{-}univuniv2\ [u, x, y] = eval\ r\text{-}phi\ [the\ (eval\ (r\text{-}universal\ 2)\ [u, u, x]), y]$
unfolding *r-univuniv2-def* **using** *assms r-univuniv2-recfn* **by** *simp*

theorem *kleene-fixed-point-theorem-2:*

assumes *recfn 2 f and total f*
shows $\exists n.$
 $recfn\ 1\ n \wedge$
 $total\ n \wedge$
 $(\forall x\ y. eval\ r\text{-}phi\ [(the\ (eval\ n\ [x])), y] = eval\ r\text{-}phi\ [(the\ (eval\ f\ [the\ (eval\ n\ [x]), x])), y])$
proof –
let $?p = encode\ r\text{-}univuniv2$
let $?s = r\text{-}smn\ 1\ 2$

```

define h where h = Cn 2 ?s [r-dummy 1 (r-const ?p), Id 2 0, Id 2 1]
then have [simp]: prim-recfn 2 h by simp
{
  fix u x y
  have eval h [u, x] = eval (Cn 2 ?s [r-dummy 1 (r-const ?p), Id 2 0, Id 2 1]) [u, x]
    using h-def by simp
  then have the (eval h [u, x]) = smn 1 ?p [u, x]
    by (simp add: r-smn)
  then have eval r-phi [the (eval h [u, x]), y] = eval r-phi [smn 1 ?p [u, x], y]
    by simp
  also have ... =
    eval r-phi
      [encode (Cn 1 (r-universal 3) (r-constn 0 ?p # r-constn 0 u # r-constn 0 x # [Id 1 0])),
        y]
    using smn[of 1 ?p [u, x]] by (simp add: numeral-3-eq-3)
  also have ... =
    eval r-phi
      [encode (Cn 1 (r-universal 3) (r-const ?p # r-const u # r-const x # [Id 1 0])), y]
      (is - = eval r-phi [encode ?f, y])
    by (simp add: r-constn-def)
  also have ... = eval ?f [y]
    using r-phi'[of ?f] by auto
  also have ... = eval (r-universal 3) [?p, u, x, y]
    using r-univuniv2-recfn r-universal r-phi by auto
  also have ... = eval r-univuniv2 [u, x, y]
    using r-universal
    by (simp add: r-universal r-univuniv2-recfn)
  finally have eval r-phi [the (eval h [u, x]), y] = eval r-univuniv2 [u, x, y] .
}
then have *: eval r-phi [the (eval h [u, x]), y] =
  eval r-phi [the (eval (r-universal 2) [u, u, x]), y]
  if eval (r-universal 2) [u, u, x] ↓ for u x y
  using r-univuniv2-converg that by simp

let ?fh = Cn 2 f [h, Id 2 1]
let ?e = encode ?fh
have recfn 2 ?fh
  using assms by simp
have total h
  by auto
then have total ?fh
  using assms Cn-total totalI2[of ?fh] by fastforce

let ?n = Cn 1 h [r-const ?e, Id 1 0]
have recfn 1 ?n
  using assms by simp
moreover have total ?n
  using ⟨total h⟩ totalI1[of ?n] by simp
moreover {
  fix x y
  have eval r-phi [(the (eval ?n [x])), y] = eval r-phi [(the (eval h [?e, x])), y]
    by simp
  also have ... = eval r-phi [the (eval (r-universal 2) [?e, ?e, x]), y]
    using * r-universal[of - 2] totalE[of ?fh 2] ⟨total ?fh⟩ ⟨recfn 2 ?fh⟩
    by (metis length-Cons list.size(3) numeral-2-eq-2)
  also have ... = eval r-phi [the (eval f [the (eval h [?e, x]), x]), y]

```

```

proof –
  have eval (r-universal 2) [?e, ?e, x] ↓
    using totalE[OF ‹total ?fh› ‹recfn 2 ‹?fh› r-universal
      by (metis length-Cons list.size(3) numeral-2-eq-2)
  moreover have eval (r-universal 2) [?e, ?e, x] = eval ?fh [?e, x]
    by (metis ‹recfn 2 ‹?fh› length-Cons list.size(3) numeral-2-eq-2 r-universal)
  then show ?thesis using assms ‹total h› by simp
qed
also have ... = eval r-phi [(the (eval f [the (eval ?n [x]), x]), y)]
  by simp
finally have eval r-phi [(the (eval ?n [x]), y)] =
  eval r-phi [(the (eval f [the (eval ?n [x]), x]), y)] .
}
ultimately show ?thesis by blast
qed

```

1.9.3 Smullyan’s double fixed-point theorem

theorem *smullyan-double-fixed-point-theorem*:

assumes *recfn* 2 *g* **and** *total g* **and** *recfn* 2 *h* **and** *total h*
shows $\exists m n$.

$(\forall x. \text{eval } r\text{-phi } [m, x] = \text{eval } r\text{-phi } [\text{the } (\text{eval } g [m, n]), x]) \wedge$
 $(\forall x. \text{eval } r\text{-phi } [n, x] = \text{eval } r\text{-phi } [\text{the } (\text{eval } h [m, n]), x])$

proof –

obtain *m* **where**

recfn 1 *m* **and**

total m **and**

m: $\forall x y. \text{eval } r\text{-phi } [\text{the } (\text{eval } m [x]), y] =$
 $\text{eval } r\text{-phi } [\text{the } (\text{eval } g [\text{the } (\text{eval } m [x]), x]), y]$

using *kleene-fixed-point-theorem-2*[*of g*] *assms*(1,2) **by** *auto*

define *k* **where** *k* = *Cn* 1 *h* [*m*, *Id* 1 0]

then have *recfn* 1 *k*

using ‹*recfn* 1 *m*› *assms*(3) **by** *simp*

have *total* (*Id* 1 0)

by (*simp* *add*: *Mn-free-imp-total*)

then have *total k*

using ‹*total m*› *assms*(4) *Cn-total k-def* ‹*recfn* 1 *k*› **by** *simp*

obtain *n* **where** *n*: $\forall x. \text{eval } r\text{-phi } [n, x] = \text{eval } r\text{-phi } [\text{the } (\text{eval } k [n]), x]$

using *rogers-fixed-point-theorem*[*of k*] ‹*recfn* 1 *k*› ‹*total k*› **by** *blast*

obtain *mm* **where** *mm*: *eval m* [*n*] ↓= *mm*

using ‹*total m*› ‹*recfn* 1 *m*› **by** *fastforce*

then have $\forall x. \text{eval } r\text{-phi } [mm, x] = \text{eval } r\text{-phi } [\text{the } (\text{eval } g [mm, n]), x]$

by (*metis* *m option.sel*)

moreover have $\forall x. \text{eval } r\text{-phi } [n, x] = \text{eval } r\text{-phi } [\text{the } (\text{eval } h [mm, n]), x]$

using *k-def* *assms*(3) ‹*total m*› ‹*recfn* 1 *m*› *mm n* **by** *simp*

ultimately show *?thesis* **by** *blast*

qed

1.10 Decidable and recursively enumerable sets

We defined *decidable* already back in Section 1.3:

decidable ?X $\equiv \exists f. \text{recfn } 1 f \wedge (\forall x. \text{eval } f [x] \downarrow = (\text{if } x \in ?X \text{ then } 1 \text{ else } 0))$

The next theorem is adapted from *halting-problem-undecidable*.

theorem *halting-problem-phi-undecidable*: \neg *decidable* $\{x. \text{eval } r\text{-phi } [x, x] \downarrow\}$
(is \neg *decidable* ?K)

proof

assume *decidable* ?K
then obtain *f* **where** *recfn* 1 *f* **and** *f*: $\forall x. \text{eval } f [x] \downarrow = (\text{if } x \in ?K \text{ then } 1 \text{ else } 0)$
using *decidable-def* **by** *auto*
define *g* **where** *g* \equiv *Cn* 1 *r-ifeq-else-diverg* [*f*, *Z*, *Z*]
then have *recfn* 1 *g*
using $\langle \text{recfn } 1 \text{ } f \rangle$ *r-ifeq-else-diverg-recfn* **by** *simp*
then obtain *i* **where** *i*: $\text{eval } r\text{-phi } [i, x] = \text{eval } g [x]$ **for** *x*
using *r-phi'* **by** *auto*
from *g-def* **have** $\text{eval } g [x] = (\text{if } x \notin ?K \text{ then } \text{Some } 0 \text{ else } \text{None})$ **for** *x*
using *r-ifeq-else-diverg-recfn* $\langle \text{recfn } 1 \text{ } f \rangle$ *f* **by** *simp*
then have $\text{eval } g [i] \downarrow \longleftrightarrow i \notin ?K$ **by** *simp*
also have $\dots \longleftrightarrow \text{eval } r\text{-phi } [i, i] \uparrow$ **by** *simp*
also have $\dots \longleftrightarrow \text{eval } g [i] \uparrow$
using *i* **by** *simp*
finally have $\text{eval } g [i] \downarrow \longleftrightarrow \text{eval } g [i] \uparrow$.
then show *False* **by** *auto*

qed

lemma *decidable-complement*: *decidable* *X* \implies *decidable* ($-$ *X*)

proof –

assume *decidable* *X*
then obtain *f* **where** *f*: *recfn* 1 *f* $\forall x. \text{eval } f [x] \downarrow = (\text{if } x \in X \text{ then } 1 \text{ else } 0)$
using *decidable-def* **by** *auto*
define *g* **where** *g* = *Cn* 1 *r-not* [*f*]
then have *recfn* 1 *g*
by (*simp* *add*: *f*(1))
moreover have $\text{eval } g [x] \downarrow = (\text{if } x \in X \text{ then } 0 \text{ else } 1)$ **for** *x*
by (*simp* *add*: *g-def* *f*)
ultimately show *?thesis* **using** *decidable-def* **by** *auto*

qed

Finite sets are decidable.

fun *r-contains* :: *nat list* \Rightarrow *recf* **where**

r-contains [] = *Z*

| *r-contains* (*x* # *xs*) = *Cn* 1 *r-ifeq* [*Id* 1 0, *r-const* *x*, *r-const* 1, *r-contains* *xs*]

lemma *r-contains-prim*: *prim-recfn* 1 (*r-contains* *xs*)

by (*induction* *xs*) *auto*

lemma *r-contains*: $\text{eval } (r\text{-contains } xs) [x] \downarrow = (\text{if } x \in \text{set } xs \text{ then } 1 \text{ else } 0)$

proof (*induction* *xs* *arbitrary*: *x*)

case *Nil*

then show *?case* **by** *simp*

next

case (*Cons* *a* *xs*)

have $\text{eval } (r\text{-contains } (a \# xs)) [x] = \text{eval } r\text{-ifeq } [x, a, 1, \text{the } (\text{eval } (r\text{-contains } xs) [x])]$

using *r-contains-prim* *prim-recfn-total* **by** *simp*

also have $\dots \downarrow = (\text{if } x = a \text{ then } 1 \text{ else if } x \in \text{set } xs \text{ then } 1 \text{ else } 0)$

using *Cons.IH* **by** *simp*

also have $\dots \downarrow = (\text{if } x = a \vee x \in \text{set } xs \text{ then } 1 \text{ else } 0)$

by *simp*

finally show *?case* **by** *simp*

qed

lemma *finite-set-decidable*: $finite\ X \implies decidable\ X$

proof –

fix $X :: nat\ set$
assume $finite\ X$
then obtain xs **where** $X = set\ xs$
using *finite-list* **by** *auto*
then have $\forall x. eval\ (r\ contains\ xs)\ [x] \downarrow = (if\ x \in X\ then\ 1\ else\ 0)$
using *r-contains* **by** *simp*
then show $decidable\ X$
using *decidable-def r-contains-prim* **by** *blast*
qed

definition *semidecidable* $:: nat\ set \Rightarrow bool$ **where**

$semidecidable\ X \equiv (\exists f. recfn\ 1\ f \wedge (\forall x. eval\ f\ [x] = (if\ x \in X\ then\ Some\ 1\ else\ None)))$

The semidecidable sets are the domains of partial recursive functions.

lemma *semidecidable-iff-domain*:

$semidecidable\ X \longleftrightarrow (\exists f. recfn\ 1\ f \wedge (\forall x. eval\ f\ [x] \downarrow \longleftrightarrow x \in X))$

proof

show $semidecidable\ X \implies \exists f. recfn\ 1\ f \wedge (\forall x. (eval\ f\ [x] \downarrow) = (x \in X))$
using *semidecidable-def* **by** (*metis option.distinct(1)*)
show $semidecidable\ X$ **if** $\exists f. recfn\ 1\ f \wedge (\forall x. (eval\ f\ [x] \downarrow) = (x \in X))$ **for** X

proof –

from that obtain f **where** $f: recfn\ 1\ f \wedge \forall x. (eval\ f\ [x] \downarrow) = (x \in X)$
by *auto*
let $?g = Cn\ 1\ (r\ const\ 1)\ [f]$
have $recfn\ 1\ ?g$
using $f(1)$ **by** *simp*
moreover have $\forall x. eval\ ?g\ [x] = (if\ x \in X\ then\ Some\ 1\ else\ None)$
using f **by** *simp*
ultimately show $semidecidable\ X$
using *semidecidable-def* **by** *blast*

qed

qed

lemma *decidable-imp-semidecidable*: $decidable\ X \implies semidecidable\ X$

proof –

assume $decidable\ X$
then obtain f **where** $f: recfn\ 1\ f \wedge \forall x. eval\ f\ [x] \downarrow = (if\ x \in X\ then\ 1\ else\ 0)$
using *decidable-def* **by** *auto*
define g **where** $g = Cn\ 1\ r\ ifeq\ else\ diverg\ [f, r\ const\ 1, r\ const\ 1]$
then have $recfn\ 1\ g$
by (*simp add: f(1)*)
have $eval\ g\ [x] = eval\ r\ ifeq\ else\ diverg\ [if\ x \in X\ then\ 1\ else\ 0, 1, 1]$ **for** x
by (*simp add: g-def f*)
then have $\bigwedge x. x \in X \implies eval\ g\ [x] \downarrow = 1$ **and** $\bigwedge x. x \notin X \implies eval\ g\ [x] \uparrow$
by *simp-all*
then show *?thesis*
using $\langle recfn\ 1\ g \rangle$ *semidecidable-def* **by** *auto*

qed

A set is recursively enumerable if it is empty or the image of a total recursive function.

definition *recursively-enumerable* $:: nat\ set \Rightarrow bool$ **where**

$recursively-enumerable\ X \equiv$

$X = \{\} \vee (\exists f. recfn\ 1\ f \wedge total\ f \wedge X = \{the\ (eval\ f\ [x]) \mid x. x \in UNIV\})$

theorem *recursively-enumerable-iff-semidecidable:*

recursively-enumerable $X \longleftrightarrow$ *semidecidable* X

proof

show *semidecidable* X **if** *recursively-enumerable* X **for** X

proof (*cases*)

assume $X = \{\}$

then show *?thesis*

using *finite-set-decidable* *decidable-imp-semidecidable*
recursively-enumerable-def *semidecidable-def*

by *blast*

next

assume $X \neq \{\}$

with that obtain f **where** f : *recfn 1 f total f* $X = \{the (eval f [x]) \mid x. x \in UNIV\}$

using *recursively-enumerable-def* **by** *blast*

define h **where** $h = Cn\ 2\ r\text{-eq}\ [Cn\ 2\ f\ [Id\ 2\ 0],\ Id\ 2\ 1]$

then have *recfn 2 h*

using $f(1)$ **by** *simp*

from h -*def* **have** h : $eval\ h\ [x,\ y] \downarrow = 0 \longleftrightarrow the\ (eval\ f\ [x]) = y$ **for** $x\ y$

using $f(1,2)$ **by** *simp*

from h -*def* \langle *recfn 2 h* \rangle *totalI2 f(2)* **have** *total h* **by** *simp*

define g **where** $g = Mn\ 1\ h$

then have *recfn 1 g*

using h -*def* $f(1)$ **by** *simp*

then have $eval\ g\ [y] =$

$(if\ (\exists x. eval\ h\ [x,\ y] \downarrow = 0 \wedge (\forall x' < x. eval\ h\ [x',\ y] \downarrow))$

$then\ Some\ (LEAST\ x. eval\ h\ [x,\ y] \downarrow = 0)$

$else\ None)$ **for** y

using g -*def* \langle *total h* \rangle $f(2)$ **by** *simp*

then have $eval\ g\ [y] =$

$(if\ \exists x. eval\ h\ [x,\ y] \downarrow = 0$

$then\ Some\ (LEAST\ x. eval\ h\ [x,\ y] \downarrow = 0)$

$else\ None)$ **for** y

using \langle *total h* \rangle \langle *recfn 2 h* \rangle **by** *simp*

then have $eval\ g\ [y] \downarrow \longleftrightarrow (\exists x. eval\ h\ [x,\ y] \downarrow = 0)$ **for** y

by *simp*

with h **have** $eval\ g\ [y] \downarrow \longleftrightarrow (\exists x. the\ (eval\ f\ [x]) = y)$ **for** y

by *simp*

with $f(3)$ **have** $eval\ g\ [y] \downarrow \longleftrightarrow y \in X$ **for** y

by *auto*

with \langle *recfn 1 g* \rangle *semidecidable-iff-domain* **show** *?thesis* **by** *auto*

qed

show *recursively-enumerable* X **if** *semidecidable* X **for** X

proof (*cases*)

assume $X = \{\}$

then show *?thesis* **using** *recursively-enumerable-def* **by** *simp*

next

assume $X \neq \{\}$

then obtain x_0 **where** $x_0 \in X$ **by** *auto*

from *that semidecidable-iff-domain* **obtain** f **where** f : *recfn 1 f* $\forall x. eval\ f\ [x] \downarrow \longleftrightarrow x \in X$

by *auto*

let $?i = encode\ f$

have i : $\bigwedge x. eval\ f\ [x] = eval\ r\text{-phi}\ [?i,\ x]$

using $r\text{-phi}'\ f(1)$ **by** *simp*

with $\langle x_0 \in X \rangle f(2)$ **have** $eval\ r\text{-phi}\ [?i,\ x_0] \downarrow$ **by** *simp*

```

then obtain  $g$  where  $g$ : recfn 1  $g$  total  $g$   $\forall x. \text{eval } r\text{-phi } [?i, x] \downarrow = (\exists y. \text{eval } g [y] \downarrow = x)$ 
  using  $f(1)$  nonempty-domain-enumerable by blast
with  $f(2)$   $i$  have  $\forall x. x \in X = (\exists y. \text{eval } g [y] \downarrow = x)$ 
  by simp
then have  $\forall x. x \in X = (\exists y. \text{the } (\text{eval } g [y]) = x)$ 
  using totalE[OF g(2) g(1)]
  by (metis One-nat-def length-Cons list.size(3) option.collapse option.sel)
then have  $X = \{\text{the } (\text{eval } g [y]) \mid y. y \in \text{UNIV}\}$ 
  by auto
with  $g(1,2)$  show ?thesis using recursively-enumerable-def by auto
qed
qed

```

The next goal is to show that a set is decidable iff. it and its complement are semidecidable. For this we use the concurrent evaluation function.

lemma *semidecidable-decidable*:

```

assumes semidecidable  $X$  and semidecidable  $(- X)$ 
shows decidable  $X$ 
proof -
obtain  $f$  where  $f$ : recfn 1  $f$   $\wedge (\forall x. \text{eval } f [x] \downarrow \longleftrightarrow x \in X)$ 
  using assms(1) semidecidable-iff-domain by auto
let  $?i = \text{encode } f$ 
obtain  $g$  where  $g$ : recfn 1  $g$   $\wedge (\forall x. \text{eval } g [x] \downarrow \longleftrightarrow x \in (- X))$ 
  using assms(2) semidecidable-iff-domain by auto
let  $?j = \text{encode } g$ 
define  $d$  where  $d = Cn$  1 r-pdec1 [Cn 1 r-parallel [r-const  $?j$ , r-const  $?i$ , Id 1 0]]
then have recfn 1  $d$ 
  by (simp add: d-def)
have  $*$ :  $\bigwedge x. \text{eval } r\text{-phi } [?i, x] = \text{eval } f [x] \wedge \bigwedge x. \text{eval } r\text{-phi } [?j, x] = \text{eval } g [x]$ 
  using  $f$   $g$  r-phi' by simp-all
have  $\text{eval } d [x] \downarrow = 1$  if  $x \in X$  for  $x$ 
proof -
  have  $\text{eval } f [x] \downarrow$ 
    using  $f$  that by simp
  moreover have  $\text{eval } g [x] \uparrow$ 
    using  $g$  that by blast
  ultimately have  $\text{eval } r\text{-parallel } [?j, ?i, x] \downarrow = \text{prod-encode } (1, \text{the } (\text{eval } f [x]))$ 
    using  $*$  r-parallel(3) by simp
  with  $d\text{-def}$  show ?thesis by simp
qed
moreover have  $\text{eval } d [x] \downarrow = 0$  if  $x \notin X$  for  $x$ 
proof -
  have  $\text{eval } g [x] \downarrow$ 
    using  $g$  that by simp
  moreover have  $\text{eval } f [x] \uparrow$ 
    using  $f$  that by blast
  ultimately have  $\text{eval } r\text{-parallel } [?j, ?i, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } g [x]))$ 
    using  $*$  r-parallel(2) by blast
  with  $d\text{-def}$  show ?thesis by simp
qed
ultimately show ?thesis
  using decidable-def  $\langle \text{recfn } 1 \ d \rangle$  by auto
qed

```

theorem *decidable-iff-semidecidable-complement*:

decidable $X \longleftrightarrow \text{semidecidable } X \wedge \text{semidecidable } (- X)$

using *semidecidable-decidable decidable-imp-semidecidable decidable-complement*
by *blast*

1.11 Rice's theorem

definition *index-set* :: *nat set* \Rightarrow *bool* **where**

index-set $I \equiv \forall i j. i \in I \wedge (\forall x. \text{eval } r\text{-phi } [i, x] = \text{eval } r\text{-phi } [j, x]) \longrightarrow j \in I$

lemma *index-set-closed-in*:

assumes *index-set* I **and** $i \in I$ **and** $\forall x. \text{eval } r\text{-phi } [i, x] = \text{eval } r\text{-phi } [j, x]$
shows $j \in I$
using *index-set-def* **assms** **by** *simp*

lemma *index-set-closed-not-in*:

assumes *index-set* I **and** $i \notin I$ **and** $\forall x. \text{eval } r\text{-phi } [i, x] = \text{eval } r\text{-phi } [j, x]$
shows $j \notin I$
using *index-set-def* **assms** **by** *metis*

theorem *rice-theorem*:

assumes *index-set* I **and** $I \neq UNIV$ **and** $I \neq \{\}$
shows $\neg \text{decidable } I$

proof

assume *decidable* I
then obtain d **where** $d: \text{recfn } 1 d \forall i. \text{eval } d [i] \downarrow = (\text{if } i \in I \text{ then } 1 \text{ else } 0)$
using *decidable-def* **by** *auto*
obtain $j_1 j_2$ **where** $j_1 \notin I$ **and** $j_2 \in I$
using *assms(2,3)* **by** *auto*
let $?if = Cn \ 2 \ r\text{-ifz } [Cn \ 2 \ d [Id \ 2 \ 0], r\text{-dummy } 1 (r\text{-const } j_2), r\text{-dummy } 1 (r\text{-const } j_1)]$
define psi **where** $psi = Cn \ 2 \ r\text{-phi } [?if, Id \ 2 \ 1]$
then have *recfn* $2 \ psi$
by (*simp add: d*)
have $\text{eval } ?if [x, y] = \text{Some } (\text{if } x \in I \text{ then } j_1 \text{ else } j_2)$ **for** $x \ y$
by (*simp add: d*)
moreover have $\text{eval } psi [x, y] = \text{eval } (Cn \ 2 \ r\text{-phi } [?if, Id \ 2 \ 1]) [x, y]$ **for** $x \ y$
using *psi-def* **by** *simp*
ultimately have $psi: \text{eval } psi [x, y] = \text{eval } r\text{-phi } [\text{if } x \in I \text{ then } j_1 \text{ else } j_2, y]$ **for** $x \ y$
by (*simp add: d*)
then have *in-I*: $\text{eval } psi [x, y] = \text{eval } r\text{-phi } [j_1, y]$ **if** $x \in I$ **for** $x \ y$
by (*simp add: that*)
have *not-in-I*: $\text{eval } psi [x, y] = \text{eval } r\text{-phi } [j_2, y]$ **if** $x \notin I$ **for** $x \ y$
by (*simp add: psi that*)
obtain n **where** $n: \forall x. \text{eval } r\text{-phi } [n, x] = \text{eval } psi [n, x]$
using *kleene-fixed-point-theorem[OF <recfn 2 psi>]* **by** *auto*
show *False*
proof *cases*
assume $n \in I$
then have $\forall x. \text{eval } r\text{-phi } [n, x] = \text{eval } r\text{-phi } [j_1, x]$
using *n in-I* **by** *simp*
then have $n \notin I$
using $\langle j_1 \notin I \rangle$ *index-set-closed-not-in[OF assms(1)]* **by** *simp*
with $\langle n \in I \rangle$ **show** *False* **by** *simp*
next
assume $n \notin I$
then have $\forall x. \text{eval } r\text{-phi } [n, x] = \text{eval } r\text{-phi } [j_2, x]$
using *n not-in-I* **by** *simp*

```

then have  $n \in I$ 
  using  $\langle j_2 \in I \rangle$  index-set-closed-in[OF assms(1)] by simp
with  $\langle n \notin I \rangle$  show False by simp
qed
qed

```

1.12 Partial recursive functions as actual functions

A well-formed *recf* describes an algorithm. Usually, however, partial recursive functions are considered to be partial functions, that is, right-unique binary relations. This distinction did not matter much until now, because we were mostly concerned with the *existence* of partial recursive functions, which is equivalent to the existence of algorithms. Whenever it did matter, we could use the extensional equivalence (\simeq). In Chapter 2, however, we will deal with sets of functions and sets of sets of functions.

For illustration consider the singleton set containing only the unary zero function. It could be expressed by $\{Z\}$, but this would not contain $Cn\ 1\ (Id\ 1\ 0)\ [Z]$, which computes the same function. The alternative representation as $\{f. f \simeq Z\}$ is not a singleton set. Another alternative would be to identify partial recursive functions with the equivalence classes of (\simeq). This would work for all arities. But since we will only need unary and binary functions, we can go for the less general but simpler alternative of regarding partial recursive functions as certain functions of types $nat \Rightarrow nat\ option$ and $nat \Rightarrow nat \Rightarrow nat\ option$. With this notation we can represent the aforementioned set by $\{\lambda-. Some\ 0\}$ and express that the function $\lambda-. Some\ 0$ is total recursive.

In addition terms get shorter, for instance, *eval r-func* $[i, x]$ becomes *func* $i\ x$.

1.12.1 The definitions

```

type-synonym partial1 =  $nat \Rightarrow nat\ option$ 

```

```

type-synonym partial2 =  $nat \Rightarrow nat \Rightarrow nat\ option$ 

```

```

definition total1 :: partial1  $\Rightarrow$  bool where
  total1  $f \equiv \forall x. f\ x \downarrow$ 

```

```

definition total2 :: partial2  $\Rightarrow$  bool where
  total2  $f \equiv \forall x\ y. f\ x\ y \downarrow$ 

```

```

lemma total1I [intro]:  $(\bigwedge x. f\ x \downarrow) \Longrightarrow total1\ f$ 
using total1-def by simp

```

```

lemma total2I [intro]:  $(\bigwedge x\ y. f\ x\ y \downarrow) \Longrightarrow total2\ f$ 
using total2-def by simp

```

```

lemma total1E [dest, simp]:  $total1\ f \Longrightarrow f\ x \downarrow$ 
using total1-def by simp

```

```

lemma total2E [dest, simp]:  $total2\ f \Longrightarrow f\ x\ y \downarrow$ 
using total2-def by simp

```

```

definition P1 :: partial1 set ( $\langle \mathcal{P} \rangle$ ) where
   $\mathcal{P} \equiv \{\lambda x. eval\ r\ [x] \mid r. recfn\ 1\ r\}$ 

```

definition $P2$:: *partial2 set* ($\langle \mathcal{P}^2 \rangle$) **where**

$\mathcal{P}^2 \equiv \{\lambda x y. \text{eval } r [x, y] \mid r. \text{recfn } 2 r\}$

definition $R1$:: *partial1 set* ($\langle \mathcal{R} \rangle$) **where**

$\mathcal{R} \equiv \{\lambda x. \text{eval } r [x] \mid r. \text{recfn } 1 r \wedge \text{total } r\}$

definition $R2$:: *partial2 set* ($\langle \mathcal{R}^2 \rangle$) **where**

$\mathcal{R}^2 \equiv \{\lambda x y. \text{eval } r [x, y] \mid r. \text{recfn } 2 r \wedge \text{total } r\}$

definition $Prim1$:: *partial1 set* **where**

$Prim1 \equiv \{\lambda x. \text{eval } r [x] \mid r. \text{prim-recfn } 1 r\}$

definition $Prim2$:: *partial2 set* **where**

$Prim2 \equiv \{\lambda x y. \text{eval } r [x, y] \mid r. \text{prim-recfn } 2 r\}$

lemma $R1\text{-imp-}P1$ [*simp, elim*]: $f \in \mathcal{R} \implies f \in \mathcal{P}$

using $R1\text{-def } P1\text{-def}$ **by** *auto*

lemma $R2\text{-imp-}P2$ [*simp, elim*]: $f \in \mathcal{R}^2 \implies f \in \mathcal{P}^2$

using $R2\text{-def } P2\text{-def}$ **by** *auto*

lemma $Prim1\text{-imp-}R1$ [*simp, elim*]: $f \in Prim1 \implies f \in \mathcal{R}$

unfolding $Prim1\text{-def } R1\text{-def}$ **by** *auto*

lemma $Prim2\text{-imp-}R2$ [*simp, elim*]: $f \in Prim2 \implies f \in \mathcal{R}^2$

unfolding $Prim2\text{-def } R2\text{-def}$ **by** *auto*

lemma $P1E$ [*elim*]:

assumes $f \in \mathcal{P}$

obtains r **where** $\text{recfn } 1 r$ **and** $\forall x. \text{eval } r [x] = f x$

using $\text{assms } P1\text{-def}$ **by** *force*

lemma $P2E$ [*elim*]:

assumes $f \in \mathcal{P}^2$

obtains r **where** $\text{recfn } 2 r$ **and** $\forall x y. \text{eval } r [x, y] = f x y$

using $\text{assms } P2\text{-def}$ **by** *force*

lemma $P1I$ [*intro*]:

assumes $\text{recfn } 1 r$ **and** $(\lambda x. \text{eval } r [x]) = f$

shows $f \in \mathcal{P}$

using $\text{assms } P1\text{-def}$ **by** *auto*

lemma $P2I$ [*intro*]:

assumes $\text{recfn } 2 r$ **and** $\bigwedge x y. \text{eval } r [x, y] = f x y$

shows $f \in \mathcal{P}^2$

proof –

have $(\lambda x y. \text{eval } r [x, y]) = f$

using $\text{assms}(2)$ **by** *simp*

then show *?thesis*

using $\text{assms}(1) P2\text{-def}$ **by** *auto*

qed

lemma $R1I$ [*intro*]:

assumes $\text{recfn } 1 r$ **and** $\text{total } r$ **and** $\bigwedge x. \text{eval } r [x] = f x$

shows $f \in \mathcal{R}$

unfolding $R1\text{-def}$

using *CollectI*[of $\lambda f. \exists r. f = (\lambda x. \text{eval } r \ [x]) \wedge \text{recfn } 1 \ r \wedge \text{total } r \ f$] *assms*
by *metis*

lemma *R1E* [*elim*]:
assumes $f \in \mathcal{R}$
obtains r **where** $\text{recfn } 1 \ r$ **and** $\text{total } r$ **and** $f = (\lambda x. \text{eval } r \ [x])$
using *assms R1-def* **by** *auto*

lemma *R2I* [*intro*]:
assumes $\text{recfn } 2 \ r$ **and** $\text{total } r$ **and** $\bigwedge x \ y. \text{eval } r \ [x, y] = f \ x \ y$
shows $f \in \mathcal{R}^2$
unfolding *R2-def*
using *CollectI*[of $\lambda f. \exists r. f = (\lambda x \ y. \text{eval } r \ [x, y]) \wedge \text{recfn } 2 \ r \wedge \text{total } r \ f$] *assms*
by *metis*

lemma *R1-SOME*:
assumes $f \in \mathcal{R}$
and $r = (\text{SOME } r'. \text{recfn } 1 \ r' \wedge \text{total } r' \wedge f = (\lambda x. \text{eval } r' \ [x]))$
(is $r = (\text{SOME } r'. \ ?P \ r')$ **)**
shows $\text{recfn } 1 \ r$
and $\bigwedge x. \text{eval } r \ [x] \downarrow$
and $\bigwedge x. f \ x = \text{eval } r \ [x]$
and $f = (\lambda x. \text{eval } r \ [x])$

proof –
obtain r' **where** $\ ?P \ r'$
using *R1E*[*OF assms(1)*] **by** *auto*
then show $\text{recfn } 1 \ r \wedge b. \text{eval } r \ [b] \downarrow \wedge \bigwedge x. f \ x = \text{eval } r \ [x]$
using *someI*[of $\ ?P \ r'$] *assms(2)* *totalE*[of r] **by** (*auto*, *metis*)
then show $f = (\lambda x. \text{eval } r \ [x])$ **by** *auto*

qed

lemma *R2E* [*elim*]:
assumes $f \in \mathcal{R}^2$
obtains r **where** $\text{recfn } 2 \ r$ **and** $\text{total } r$ **and** $f = (\lambda x_1 \ x_2. \text{eval } r \ [x_1, x_2])$
using *assms R2-def* **by** *auto*

lemma *R1-imp-total1* [*simp*]: $f \in \mathcal{R} \implies \text{total1 } f$
using *total1I* **by** *fastforce*

lemma *R2-imp-total2* [*simp*]: $f \in \mathcal{R}^2 \implies \text{total2 } f$
using *totalE* **by** *fastforce*

lemma *Prim1I* [*intro*]:
assumes $\text{prim-recfn } 1 \ r$ **and** $\bigwedge x. f \ x = \text{eval } r \ [x]$
shows $f \in \text{Prim1}$
using *assms Prim1-def* **by** *blast*

lemma *Prim2I* [*intro*]:
assumes $\text{prim-recfn } 2 \ r$ **and** $\bigwedge x \ y. f \ x \ y = \text{eval } r \ [x, y]$
shows $f \in \text{Prim2}$
using *assms Prim2-def* **by** *blast*

lemma *P1-total-imp-R1* [*intro*]:
assumes $f \in \mathcal{P}$ **and** $\text{total1 } f$
shows $f \in \mathcal{R}$
using *assms total1I* **by** *force*

lemma *P2-total-imp-R2* [intro]:
assumes $f \in \mathcal{P}^2$ **and** *total2* f
shows $f \in \mathcal{R}^2$
using *assms totalI2* **by** *force*

1.12.2 Some simple properties

In order to show that a *partial1* or *partial2* function is in \mathcal{P} , \mathcal{P}^2 , \mathcal{R} , \mathcal{R}^2 , *Prim1*, or *Prim2* we will usually have to find a suitable *recf*. But for some simple or frequent cases this section provides shortcuts.

lemma *identity-in-R1*: *Some* $\in \mathcal{R}$

proof –

have $\forall x. \text{eval } (Id\ 1\ 0)\ [x] \Downarrow = x$ **by** *simp*
moreover **have** *recfn* 1 $(Id\ 1\ 0)$ **by** *simp*
moreover **have** *total* $(Id\ 1\ 0)$
by (*simp add: totalI1*)
ultimately **show** *?thesis* **by** *blast*

qed

lemma *P2-proj-P1* [*simp, elim*]:

assumes $\psi \in \mathcal{P}^2$
shows $\psi\ i \in \mathcal{P}$

proof –

from *assms* **obtain** u **where** $u: \text{recfn}\ 2\ u\ (\lambda x_1\ x_2. \text{eval } u\ [x_1, x_2]) = \psi$
by *auto*
define v **where** $v \equiv Cn\ 1\ u\ [r\text{-const } i, Id\ 1\ 0]$
then **have** *recfn* 1 $v\ (\lambda x. \text{eval } v\ [x]) = \psi\ i$
using u **by** *auto*
then **show** *?thesis* **by** *auto*

qed

lemma *R2-proj-R1* [*simp, elim*]:

assumes $\psi \in \mathcal{R}^2$
shows $\psi\ i \in \mathcal{R}$

proof –

from *assms* **have** $\psi \in \mathcal{P}^2$ **by** *simp*
then **have** $\psi\ i \in \mathcal{P}$ **by** *auto*
moreover **have** *total1* $(\psi\ i)$
using *assms* **by** (*simp add: totalI1*)
ultimately **show** *?thesis* **by** *auto*

qed

lemma *const-in-Prim1*: $(\lambda-. \text{Some } c) \in \text{Prim1}$

proof –

define r **where** $r = r\text{-const } c$
then **have** $\bigwedge x. \text{eval } r\ [x] = \text{Some } c$ **by** *simp*
moreover **have** *recfn* 1 r *Mn-free* r
using *r-def* **by** *simp-all*
ultimately **show** *?thesis* **by** *auto*

qed

lemma *concat-P1-P1*:

assumes $f \in \mathcal{P}$ **and** $g \in \mathcal{P}$
shows $(\lambda x. \text{if } g\ x \Downarrow \wedge f\ (\text{the } (g\ x)) \Downarrow \text{ then } \text{Some } (\text{the } (f\ (\text{the } (g\ x)))) \text{ else } \text{None}) \in \mathcal{P}$

(is ?h ∈ \mathcal{P})

proof –

obtain *rf* **where** *rf*: *recfn* 1 *rf* $\forall x. \text{eval } rf [x] = f x$
using *assms*(1) **by** *auto*

obtain *rg* **where** *rg*: *recfn* 1 *rg* $\forall x. \text{eval } rg [x] = g x$
using *assms*(2) **by** *auto*

let ?rh = *Cn* 1 *rf* [*rg*]
have *recfn* 1 ?rh
using *rf*(1) *rg*(1) **by** *simp*

moreover **have** *eval* ?rh [x] = ?h x **for** x
using *rf* *rg* **by** *simp*

ultimately **show** ?thesis **by** *blast*

qed

lemma *P1-update-P1*:

assumes *f* ∈ \mathcal{P}
shows *f*(*x*:=*z*) ∈ \mathcal{P}

proof (*cases* *z*)

case *None*

define *re* **where** *re* ≡ *Mn* 1 (*r-constn* 1 1)
from *assms* **obtain** *r* **where** *r*: *recfn* 1 *r* ($\lambda u. \text{eval } r [u] = f$)
by *auto*

define *r'* **where** *r'* = *Cn* 1 (*r-lifz* *re* *r*) [*Cn* 1 *r-eq* [*Id* 1 0, *r-const* *x*], *Id* 1 0]
have *recfn* 1 *r'*
using *r*(1) *r'-def* *re-def* **by** *simp*

then **have** *eval* *r'* [u] = *eval* (*r-lifz* *re* *r*) [*if* *u* = *x* *then* 0 *else* 1, u] **for** *u*
using *r'-def* **by** *simp*

with *r*(1) **have** *eval* *r'* [u] = (*if* *u* = *x* *then* *None* *else* *eval* *r* [u]) **for** *u*
using *re-def* *re-def* **by** *simp*

with *r*(2) **have** *eval* *r'* [u] = (*f*(*x*:=*None*)) *u* **for** *u*
by *auto*

then **have** ($\lambda u. \text{eval } r' [u] = f(x:=None)$)
by *auto*

with *None* <*recfn* 1 *r'*> **show** ?thesis **by** *auto*

next

case (*Some* *y*)

from *assms* **obtain** *r* **where** *r*: *recfn* 1 *r* ($\lambda u. \text{eval } r [u] = f$)
by *auto*

define *r'* **where**
r' ≡ *Cn* 1 (*r-lifz* (*r-const* *y*) *r*) [*Cn* 1 *r-eq* [*Id* 1 0, *r-const* *x*], *Id* 1 0]
have *recfn* 1 *r'*
using *r*(1) *r'-def* **by** *simp*

then **have** *eval* *r'* [u] = *eval* (*r-lifz* (*r-const* *y*) *r*) [*if* *u* = *x* *then* 0 *else* 1, u] **for** *u*
using *r'-def* **by** *simp*

with *r*(1) **have** *eval* *r'* [u] = (*if* *u* = *x* *then* *Some* *y* *else* *eval* *r* [u]) **for** *u*
by *simp*

with *r*(2) **have** *eval* *r'* [u] = (*f*(*x*:=*Some* *y*)) *u* **for** *u*
by *auto*

then **have** ($\lambda u. \text{eval } r' [u] = f(x:=Some y)$)
by *auto*

with *Some* <*recfn* 1 *r'*> **show** ?thesis **by** *auto*

qed

lemma *swap-P2*:

assumes *f* ∈ \mathcal{P}^2
shows ($\lambda x y. f y x$) ∈ \mathcal{P}^2

proof –
obtain r **where** $r: \text{recfn } 2 \ r \ \wedge x \ y. \text{ eval } r \ [x, y] = f \ x \ y$
using assms **by** auto
then have $\text{eval } (r\text{-swap } r) \ [x, y] = f \ y \ x$ **for** $x \ y$
by simp
moreover have $\text{recfn } 2 \ (r\text{-swap } r)$
using $r\text{-swap-recfn } r(1)$ **by** simp
ultimately show $?thesis$ **by** auto
qed

lemma swap-R2 :
assumes $f \in \mathcal{R}^2$
shows $(\lambda x \ y. f \ y \ x) \in \mathcal{R}^2$
using $\text{swap-P2[of } f] \ \text{assms}$
by $(\text{meson } P2\text{-total-imp-R2 } R2\text{-imp-P2 } R2\text{-imp-total2 } \text{total2E } \text{total2I})$

lemma skip-P1 :
assumes $f \in \mathcal{P}$
shows $(\lambda x. f \ (x + n)) \in \mathcal{P}$

proof –
obtain r **where** $r: \text{recfn } 1 \ r \ \wedge x. \text{ eval } r \ [x] = f \ x$
using assms **by** auto
let $?s = Cn \ 1 \ r \ [Cn \ 1 \ r\text{-add } [Id \ 1 \ 0, r\text{-const } n]]$
have $\text{recfn } 1 \ ?s$
using r **by** simp
have $\text{eval } ?s \ [x] = \text{eval } r \ [x + n]$ **for** x
using r **by** simp
with r **have** $\text{eval } ?s \ [x] = f \ (x + n)$ **for** x
by simp
with $\langle \text{recfn } 1 \ ?s \rangle$ **show** $?thesis$ **by** blast
qed

lemma skip-R1 :
assumes $f \in \mathcal{R}$
shows $(\lambda x. f \ (x + n)) \in \mathcal{R}$
using $\text{assms } \text{skip-P1 } R1\text{-imp-total1 } \text{total1-def}$ **by** auto

1.12.3 The Gödel numbering φ

While the term *Gödel numbering* is often used generically for mappings between natural numbers and mathematical concepts, the inductive inference literature uses it in a more specific sense. There it is equivalent to the notion of acceptable numbering [12]: For every numbering there is a recursive function mapping the numbering's indices to equivalent ones of a Gödel numbering.

definition $\text{goedel-numbering} :: \text{partial2} \Rightarrow \text{bool}$ **where**
 $\text{goedel-numbering } \psi \equiv \psi \in \mathcal{P}^2 \ \wedge \ (\forall \chi \in \mathcal{P}^2. \exists c \in \mathcal{R}. \forall i. \chi \ i = \psi \ (\text{the } (c \ i)))$

lemma $\text{goedel-numbering-P2}$:
assumes $\text{goedel-numbering } \psi$
shows $\psi \in \mathcal{P}^2$
using $\text{goedel-numbering-def } \text{assms}$ **by** simp

lemma goedel-numberingE :
assumes $\text{goedel-numbering } \psi$ **and** $\chi \in \mathcal{P}^2$
obtains c **where** $c \in \mathcal{R}$ **and** $\forall i. \chi \ i = \psi \ (\text{the } (c \ i))$

using *assms goedel-numbering-def* by *blast*

lemma *goedel-numbering-universal*:

assumes *goedel-numbering* ψ **and** $f \in \mathcal{P}$

shows $\exists i. \psi i = f$

proof –

define $\chi :: \text{partial2}$ **where** $\chi = (\lambda i. f)$

have $\chi \in \mathcal{P}^2$

proof –

obtain *rf* **where** *rf*: *recfn 1 rf* $\wedge x. \text{eval } rf [x] = f x$

using *assms(2)* **by** *auto*

define *r* **where** $r = \text{Cn } 2 \text{ rf } [Id \ 2 \ 1]$

then have *r*: *recfn 2 r* $\wedge i x. \text{eval } r [i, x] = \text{eval } rf [x]$

using *rf(1)* **by** *simp-all*

with *rf(2)* **have** $\wedge i x. \text{eval } r [i, x] = f x$ **by** *simp*

with *r(1)* **show** *?thesis* **using** χ -*def* **by** *auto*

qed

then obtain *c* **where** $c \in \mathcal{R}$ **and** $\forall i. \chi i = \psi$ (*the (c i)*)

using *goedel-numbering-def* *assms(1)* **by** *auto*

with χ -*def* **show** *?thesis* **by** *auto*

qed

Our standard Gödel numbering is based on *r-phi*:

definition *phi* :: *partial2* ($\langle \varphi \rangle$) **where**

$\varphi i x \equiv \text{eval } r\text{-phi } [i, x]$

lemma *phi-in-P2*: $\varphi \in \mathcal{P}^2$

unfolding *phi-def* **using** *r-phi-recfn* **by** *blast*

Indices of any numbering can be translated into equivalent indices of φ , which thus is a Gödel numbering.

lemma *numbering-translation-for-phi*:

assumes $\psi \in \mathcal{P}^2$

shows $\exists c \in \mathcal{R}. \forall i. \psi i = \varphi$ (*the (c i)*)

proof –

obtain *psi* **where** *psi*: *recfn 2 psi* $\wedge i x. \text{eval } psi [i, x] = \psi i x$

using *assms* **by** *auto*

with *numbering-translation* **obtain** *b* **where**

recfn 1 b total b $\forall i x. \text{eval } psi [i, x] = \text{eval } r\text{-phi } [the (eval b [i]), x]$

by *blast*

moreover from this **obtain** *c* **where** $c: c \in \mathcal{R} \forall i. c i = \text{eval } b [i]$

by *fast*

ultimately have $\psi i x = \varphi$ (*the (c i)*) *x* **for** *i x*

using *phi-def* *psi(2)* **by** *presburger*

then have $\psi i = \varphi$ (*the (c i)*) **for** *i*

by *auto*

then show *?thesis* **using** *c(1)* **by** *blast*

qed

corollary *goedel-numbering-phi*: *goedel-numbering* φ

unfolding *goedel-numbering-def* **using** *numbering-translation-for-phi* *phi-in-P2* **by** *simp*

corollary *phi-universal*:

assumes $f \in \mathcal{P}$

obtains *i* **where** $\varphi i = f$

using *goedel-numbering-universal*[*OF goedel-numbering-phi* *assms*] **by** *auto*

1.12.4 Fixed-point theorems

The fixed-point theorems look somewhat cleaner in the new notation. We will only need the following ones in the next chapter.

theorem *kleene-fixed-point*:

fixes $k :: \text{nat}$

assumes $\psi \in \mathcal{P}^2$

obtains i **where** $i \geq k$ **and** $\varphi i = \psi i$

proof –

obtain $r\text{-psi}$ **where** $r\text{-psi}$: $\text{recfn } 2 \text{ } r\text{-psi} \wedge i x. \text{eval } r\text{-psi } [i, x] = \psi i x$
using *assms* **by** *auto*

then obtain i **where** $i: i \geq k \forall x. \text{eval } r\text{-psi } [i, x] = \text{eval } r\text{-psi } [i, x]$
using *kleene-fixed-point-theorem* **by** *blast*

then have $\forall x. \varphi i x = \psi i x$

using *phi-def r-psi* **by** *simp*

then show *?thesis* **using** i **that** **by** *blast*

qed

theorem *smullyan-double-fixed-point*:

assumes $g \in \mathcal{R}^2$ **and** $h \in \mathcal{R}^2$

obtains $m n$ **where** $\varphi m = \varphi (\text{the } (g m n))$ **and** $\varphi n = \varphi (\text{the } (h m n))$

proof –

obtain rg **where** rg : $\text{recfn } 2 \text{ } rg \text{ total } rg \ g = (\lambda x y. \text{eval } rg \ [x, y])$

using *R2E[OF assms(1)]* **by** *auto*

moreover obtain rh **where** rh : $\text{recfn } 2 \text{ } rh \text{ total } rh \ h = (\lambda x y. \text{eval } rh \ [x, y])$

using *R2E[OF assms(2)]* **by** *auto*

ultimately obtain $m n$ **where**

$\forall x. \text{eval } r\text{-phi } [m, x] = \text{eval } r\text{-phi } [\text{the } (\text{eval } rg \ [m, n]), x]$

$\forall x. \text{eval } r\text{-phi } [n, x] = \text{eval } r\text{-phi } [\text{the } (\text{eval } rh \ [m, n]), x]$

using *smullyan-double-fixed-point-theorem[of rg rh]* **by** *blast*

then have $\varphi m = \varphi (\text{the } (g m n))$ **and** $\varphi n = \varphi (\text{the } (h m n))$

using *phi-def rg rh* **by** *auto*

then show *?thesis* **using** $that$ **by** *simp*

qed

end

Chapter 2

Inductive inference of recursive functions

```
theory Inductive-Inference-Basics
  imports Standard-Results
begin
```

Inductive inference originates from work by Solomonoff [13, 14] and Gold [9, 8] and comes in many variations. The common theme is to infer additional information about objects, such as formal languages or functions, from incomplete data, such as finitely many words contained in the language or argument-value pairs of the function. Often-times “additional information” means complete information, such that the task becomes identification of the object.

The basic setting in inductive inference of recursive functions is as follows. Let us denote, for a total function f , by f^n the code of the list $[f(0), \dots, f(n)]$. Let U be a set (called *class*) of total recursive functions, and ψ a binary partial recursive function (called *hypothesis space*). A partial recursive function S (called *strategy*) is said to *learn* U *in the limit with respect to* ψ if for all $f \in U$,

- the value $S(f^n)$ is defined for all $n \in \mathbb{N}$,
- the sequence $S(f^0), S(f^1), \dots$ converges to an $i \in \mathbb{N}$ with $\psi_i = f$.

Both the output $S(f^n)$ of the strategy and its interpretation as a function $\psi_{S(f^n)}$ are called *hypothesis*. The set of all classes learnable in the limit by S with respect to ψ is denoted by $\text{LIM}_\psi(S)$. Moreover we set $\text{LIM}_\psi = \bigcup_{S \in \mathcal{P}} \text{LIM}_\psi(S)$ and $\text{LIM} = \bigcup_{\psi \in \mathcal{P}^2} \text{LIM}_\psi$. We call the latter set the *inference type* LIM.

Many aspects of this setting can be varied. We shall consider:

- Intermediate hypotheses: $\psi_{S(f^n)}$ can be required to be total or to be in the class U , or to coincide with f on arguments up to n , or a myriad of other conditions or combinations thereof.
- Convergence of hypotheses:
 - The strategy can be required to output not a sequence but a single hypothesis, which must be correct.
 - The strategy can be required to converge to a *function* rather than an index.

We formalize five kinds of results (\mathcal{I} and \mathcal{I}' stand for inference types):

- Comparison of learning power: results of the form $\mathcal{I} \subset \mathcal{I}'$, in particular showing that the inclusion is proper (Sections 2.3, 2.4, 2.5, 2.6, 2.7, 2.9, 2.10, 2.11).
- Whether \mathcal{I} is closed under the subset relation: $U \in \mathcal{I} \wedge V \subseteq U \implies V \in \mathcal{I}$.
- Whether \mathcal{I} is closed under union: $U \in \mathcal{I} \wedge V \in \mathcal{I} \implies U \cup V \in \mathcal{I}$ (Section 2.12).
- Whether every class in \mathcal{I} can be learned with respect to a Gödel numbering as hypothesis space (Section 2.2).
- Whether every class in \mathcal{I} can be learned by a *total* recursive strategy (Section 2.8).

The bulk of this chapter is devoted to the first category of results. Most results that we are going to formalize have been called “classical” by Jantke and Beick [10], who compare a large number of inference types. Another comparison is by Case and Smith [6]. Angluin and Smith [1] give an overview of various forms of inductive inference.

All (interesting) proofs herein are based on my lecture notes of the *Induktive Inferenz* lectures by Rolf Wiehagen from 1999/2000 and 2000/2001 at the University of Kaiserslautern. I have given references to the original proofs whenever I was able to find them. For the other proofs, as well as for those that I had to contort beyond recognition, I provide proof sketches.

2.1 Preliminaries

Throughout the chapter, in particular in proof sketches, we use the following notation.

Let $b \in \mathbb{N}^*$ be a list of numbers. We write $|b|$ for its length and b_i for the i -th element ($i = 0, \dots, |b| - 1$). Concatenation of numbers and lists works in the obvious way; for instance, jbk with $j, k \in \mathbb{N}$, $b \in \mathbb{N}^*$ refers to the list $jb_0 \dots b_{|b|-1}k$. For $0 \leq i < |b|$, the term $b_{i:=v}$ denotes the list $b_0 \dots b_{i-1}vb_{i+1} \dots b_{|b|-1}$. The notation $b_{<i}$ refers to $b_0 \dots b_{i-1}$ for $0 < i \leq |b|$. Moreover, v^n is short for the list consisting of n times the value $v \in \mathbb{N}$.

Unary partial functions can be regarded as infinite sequences consisting of numbers and the symbol \uparrow denoting undefinedness. We abbreviate the empty function by \uparrow^∞ and the constant zero function by 0^∞ . A function can be written as a list concatenated with a partial function. For example, $jb \uparrow^\infty$ is the function

$$x \mapsto \begin{cases} j & \text{if } x = 0, \\ b_{x-1} & \text{if } 0 < x \leq |b|, \\ \uparrow & \text{otherwise,} \end{cases}$$

and jp , where p is a function, means

$$x \mapsto \begin{cases} j & \text{if } x = 0, \\ p(x-1) & \text{otherwise.} \end{cases}$$

A *numbering* is a function $\psi \in \mathcal{P}^2$.

2.1.1 The prefixes of a function

A *prefix*, also called *initial segment*, is a list of initial values of a function.

definition *prefix* :: *partial1* \Rightarrow *nat* \Rightarrow *nat list* **where**

$\text{prefix } f \ n \equiv \text{map } (\lambda x. \text{the } (f \ x)) \ [0..<\text{Suc } n]$

lemma *length-prefix* [*simp*]: $\text{length } (\text{prefix } f \ n) = \text{Suc } n$
unfolding *prefix-def* **by** *simp*

lemma *prefix-nth* [*simp*]:
assumes $k < \text{Suc } n$
shows $\text{prefix } f \ n \ ! \ k = \text{the } (f \ k)$
unfolding *prefix-def* **using** *assms nth-map-upt*[*of* $k \ \text{Suc } n \ 0 \ \lambda x. \text{the } (f \ x)$] **by** *simp*

lemma *prefixI*:
assumes $\text{length } vs > 0$ **and** $\bigwedge x. x < \text{length } vs \implies f \ x \downarrow = vs \ ! \ x$
shows $\text{prefix } f \ (\text{length } vs - 1) = vs$
using *assms nth-equalityI*[*of* $\text{prefix } f \ (\text{length } vs - 1) \ vs$] **by** *simp*

lemma *prefixI'*:
assumes $\text{length } vs = \text{Suc } n$ **and** $\bigwedge x. x < \text{Suc } n \implies f \ x \downarrow = vs \ ! \ x$
shows $\text{prefix } f \ n = vs$
using *assms nth-equalityI*[*of* $\text{prefix } f \ (\text{length } vs - 1) \ vs$] **by** *simp*

lemma *prefixE*:
assumes $\text{prefix } f \ (\text{length } vs - 1) = vs$
and $f \in \mathcal{R}$
and $\text{length } vs > 0$
and $x < \text{length } vs$
shows $f \ x \downarrow = vs \ ! \ x$
using *assms length-prefix prefix-nth*[*of* $x \ \text{length } vs - 1 \ f$] **by** *simp*

lemma *prefix-eqI*:
assumes $\bigwedge x. x \leq n \implies f \ x = g \ x$
shows $\text{prefix } f \ n = \text{prefix } g \ n$
using *assms prefix-def* **by** *simp*

lemma *prefix-0*: $\text{prefix } f \ 0 = [\text{the } (f \ 0)]$
using *prefix-def* **by** *simp*

lemma *prefix-Suc*: $\text{prefix } f \ (\text{Suc } n) = \text{prefix } f \ n \ @ \ [\text{the } (f \ (\text{Suc } n))]$
unfolding *prefix-def* **by** *simp*

lemma *take-prefix*:
assumes $f \in \mathcal{R}$ **and** $k \leq n$
shows $\text{prefix } f \ k = \text{take } (\text{Suc } k) \ (\text{prefix } f \ n)$
proof –
let $?vs = \text{take } (\text{Suc } k) \ (\text{prefix } f \ n)$
have $\text{length } ?vs = \text{Suc } k$
using *assms*(2) **by** *simp*
then have $\bigwedge x. x < \text{length } ?vs \implies f \ x \downarrow = ?vs \ ! \ x$
using *assms* **by** *auto*
then show *thesis*
using *prefixI*[**where** $?vs = ?vs$] $\langle \text{length } ?vs = \text{Suc } k \rangle$ **by** *simp*
qed

Strategies receive prefixes in the form of encoded lists. The term “prefix” refers to both encoded and unencoded lists. We use the notation $f \triangleright n$ for the prefix f^n .

definition *init* :: *partial1* \Rightarrow *nat* \Rightarrow *nat* (**infix** $\langle \triangleright \rangle$ 110) **where**
 $f \triangleright n \equiv \text{list-encode } (\text{prefix } f \ n)$

lemma *init-neq-zero*: $f \triangleright n \neq 0$

unfolding *init-def prefix-def* **using** *list-encode-0* **by** *fastforce*

lemma *init-prefixE* [*elim*]: $\text{prefix } f \ n = \text{prefix } g \ n \implies f \triangleright n = g \triangleright n$

unfolding *init-def* **by** *simp*

lemma *init-eqI*:

assumes $\bigwedge x. x \leq n \implies f \ x = g \ x$

shows $f \triangleright n = g \triangleright n$

unfolding *init-def* **using** *prefix-eqI* [*OF assms*] **by** *simp*

lemma *initI*:

assumes $e\text{-length } e > 0$ **and** $\bigwedge x. x < e\text{-length } e \implies f \ x \downarrow = e\text{-nth } e \ x$

shows $f \triangleright (e\text{-length } e - 1) = e$

unfolding *init-def* **using** *assms prefixI* **by** *simp*

lemma *initI'*:

assumes $e\text{-length } e = \text{Suc } n$ **and** $\bigwedge x. x < \text{Suc } n \implies f \ x \downarrow = e\text{-nth } e \ x$

shows $f \triangleright n = e$

unfolding *init-def* **using** *assms prefixI'* **by** *simp*

lemma *init-iff-list-eq-upto*:

assumes $f \in \mathcal{R}$ **and** $e\text{-length } vs > 0$

shows $(\forall x < e\text{-length } vs. f \ x \downarrow = e\text{-nth } vs \ x) \iff \text{prefix } f \ (e\text{-length } vs - 1) = \text{list-decode } vs$

using *prefixI* [*OF assms(2)*] *prefixE* [*OF - assms*] **by** *auto*

lemma *length-init* [*simp*]: $e\text{-length } (f \triangleright n) = \text{Suc } n$

unfolding *init-def* **by** *simp*

lemma *init-Suc-snoc*: $f \triangleright (\text{Suc } n) = e\text{-snoc } (f \triangleright n) \text{ (the } (f \ (\text{Suc } n)))$

unfolding *init-def* **by** (*simp add: prefix-Suc*)

lemma *nth-init*: $i < \text{Suc } n \implies e\text{-nth } (f \triangleright n) \ i = \text{the } (f \ i)$

unfolding *init-def* **using** *prefix-nth* **by** *auto*

lemma *hd-init* [*simp*]: $e\text{-hd } (f \triangleright n) = \text{the } (f \ 0)$

unfolding *init-def* **using** *init-neq-zero* **by** (*simp add: e-hd-nth0*)

lemma *list-decode-init* [*simp*]: $\text{list-decode } (f \triangleright n) = \text{prefix } f \ n$

unfolding *init-def* **by** *simp*

lemma *init-eq-iff-eq-upto*:

assumes $g \in \mathcal{R}$ **and** $f \in \mathcal{R}$

shows $(\forall j < \text{Suc } n. g \ j = f \ j) \iff g \triangleright n = f \triangleright n$

using *assms initI' init-iff-list-eq-upto length-init list-decode-init*

by (*metis diff-Suc-1 zero-less-Suc*)

definition *is-init-of* :: $\text{nat} \Rightarrow \text{partial1} \Rightarrow \text{bool}$ **where**

$\text{is-init-of } t \ f \equiv \forall i < e\text{-length } t. f \ i \downarrow = e\text{-nth } t \ i$

lemma *not-initial-imp-not-eq*:

assumes $\bigwedge x. x < \text{Suc } n \implies f \ x \downarrow$ **and** $\neg (\text{is-init-of } (f \triangleright n) \ g)$

shows $f \neq g$

using *is-init-of-def assms* **by** *auto*

lemma *all-init-eq-imp-fun-eq*:
 assumes $f \in \mathcal{R}$ and $g \in \mathcal{R}$ and $\bigwedge n. f \triangleright n = g \triangleright n$
 shows $f = g$
proof
 fix n
 from *assms* have $\text{prefix } f \ n = \text{prefix } g \ n$
 by (*metis* *init-def* *list-decode-encode*)
 then have $\text{the } (f \ n) = \text{the } (g \ n)$
 unfolding *init-def* *prefix-def* by *simp*
 then show $f \ n = g \ n$
 using *assms*(1,2) by (*meson* *R1-imp-total1* *option.expand* *total1E*)
qed

corollary *neq-fun-neq-init*:
 assumes $f \in \mathcal{R}$ and $g \in \mathcal{R}$ and $f \neq g$
 shows $\exists n. f \triangleright n \neq g \triangleright n$
 using *assms* *all-init-eq-imp-fun-eq* by *auto*

lemma *eq-init-forall-le*:
 assumes $f \triangleright n = g \triangleright n$ and $m \leq n$
 shows $f \triangleright m = g \triangleright m$
proof –
 from *assms*(1) have $\text{prefix } f \ n = \text{prefix } g \ n$
 by (*metis* *init-def* *list-decode-encode*)
 then have $\text{the } (f \ k) = \text{the } (g \ k)$ if $k \leq n$ for k
 using *prefix-def* that by *auto*
 then have $\text{the } (f \ k) = \text{the } (g \ k)$ if $k \leq m$ for k
 using *assms*(2) that by *simp*
 then have $\text{prefix } f \ m = \text{prefix } g \ m$
 using *prefix-def* by *simp*
 then show *?thesis* by (*simp* *add: init-def*)
qed

corollary *neq-init-forall-ge*:
 assumes $f \triangleright n \neq g \triangleright n$ and $m \geq n$
 shows $f \triangleright m \neq g \triangleright m$
 using *eq-init-forall-le* *assms* by *blast*

lemma *e-take-init*:
 assumes $f \in \mathcal{R}$ and $k < \text{Suc } n$
 shows $e\text{-take } (\text{Suc } k) (f \triangleright n) = f \triangleright k$
 using *assms* *take-prefix* by (*simp* *add: init-def* *less-Suc-eq-le*)

lemma *init-butlast-init*:
 assumes *total1* f and $f \triangleright n = e$ and $n > 0$
 shows $f \triangleright (n - 1) = e\text{-butlast } e$
proof –
 let $?e = e\text{-butlast } e$
 have $e\text{-length } e = \text{Suc } n$
 using *assms*(2) by *auto*
 then have $\text{len: } e\text{-length } ?e = n$
 by *simp*
 have $f \triangleright (e\text{-length } ?e - 1) = ?e$
proof (*rule* *initI*)
 show $0 < e\text{-length } ?e$
 using *assms*(3) *len* by *simp*

```

have  $\bigwedge x. x < e\text{-length } e \implies f\ x \downarrow = e\text{-nth } e\ x$ 
  using assms(1,2) total1-def  $\langle e\text{-length } e = \text{Suc } n \rangle$  by auto
then show  $\bigwedge x. x < e\text{-length } ?e \implies f\ x \downarrow = e\text{-nth } ?e\ x$ 
  by (simp add: butlast-conv-take)
qed
with len show ?thesis by simp
qed

```

Some definitions make use of recursive predicates, that is, 01-valued functions.

definition *RPred1* :: *partial1 set* $\langle \mathcal{R}_{01} \rangle$ **where**
 $\mathcal{R}_{01} \equiv \{f. f \in \mathcal{R} \wedge (\forall x. f\ x \downarrow = 0 \vee f\ x \downarrow = 1)\}$

lemma *RPred1-subseteq-R1*: $\mathcal{R}_{01} \subseteq \mathcal{R}$
unfolding *RPred1-def* **by** *auto*

lemma *const0-in-RPred1*: $(\lambda-. \text{Some } 0) \in \mathcal{R}_{01}$
using *RPred1-def const-in-Prim1* **by** *fast*

lemma *RPred1-altdef*: $\mathcal{R}_{01} = \{f. f \in \mathcal{R} \wedge (\forall x. \text{the } (f\ x) \leq 1)\}$
(is $\mathcal{R}_{01} = ?S$ **)**

proof
show $\mathcal{R}_{01} \subseteq ?S$
proof
fix *f*
assume *f*: $f \in \mathcal{R}_{01}$
with *RPred1-def* **have** $f \in \mathcal{R}$ **by** *auto*
from *f* **have** $\forall x. f\ x \downarrow = 0 \vee f\ x \downarrow = 1$
by (*simp add: RPred1-def*)
then have $\forall x. \text{the } (f\ x) \leq 1$
by (*metis eq-refl less-Suc-eq-le zero-less-Suc option.sel*)
with $\langle f \in \mathcal{R} \rangle$ **show** $f \in ?S$ **by** *simp*

qed
show $?S \subseteq \mathcal{R}_{01}$
proof
fix *f*
assume *f*: $f \in ?S$
then have $f \in \mathcal{R}$ **by** *simp*
then have *total*: $\bigwedge x. f\ x \downarrow$ **by** *auto*
from *f* **have** $\forall x. \text{the } (f\ x) = 0 \vee \text{the } (f\ x) = 1$
by (*simp add: le-eq-less-or-eq*)
with *total* **have** $\forall x. f\ x \downarrow = 0 \vee f\ x \downarrow = 1$
by (*metis option.collapse*)
then show $f \in \mathcal{R}_{01}$
using $\langle f \in \mathcal{R} \rangle$ *RPred1-def* **by** *auto*
qed
qed

2.1.2 NUM

A class of recursive functions is in NUM if it can be embedded in a total numbering. Thus, for learning such classes there is always a total hypothesis space available.

definition *NUM* :: *partial1 set set* **where**
 $\text{NUM} \equiv \{U. \exists \psi \in \mathcal{R}^2. \forall f \in U. \exists i. \psi\ i = f\}$

definition *NUM-wrt* :: *partial2* \Rightarrow *partial1 set set* **where**

$\psi \in \mathcal{R}^2 \implies \text{NUM-wrt } \psi \equiv \{U. \forall f \in U. \exists i. \psi \ i = f\}$

lemma *NUM-I* [*intro*]:
assumes $\psi \in \mathcal{R}^2$ **and** $\bigwedge f. f \in U \implies \exists i. \psi \ i = f$
shows $U \in \text{NUM}$
using *assms NUM-def* **by** *blast*

lemma *NUM-E* [*dest*]:
assumes $U \in \text{NUM}$
shows $U \subseteq \mathcal{R}$
and $\exists \psi \in \mathcal{R}^2. \forall f \in U. \exists i. \psi \ i = f$
using *NUM-def assms* **by** (*force, auto*)

lemma *NUM-closed-subseteq*:
assumes $U \in \text{NUM}$ **and** $V \subseteq U$
shows $V \in \text{NUM}$
using *assms subset-eq*[*of V U*] *NUM-I* **by** *auto*

This is the classical diagonalization proof showing that there is no total numbering containing all total recursive functions.

lemma *R1-not-in-NUM*: $\mathcal{R} \notin \text{NUM}$

proof

assume $\mathcal{R} \in \text{NUM}$
then obtain ψ **where** *num*: $\psi \in \mathcal{R}^2 \ \forall f \in \mathcal{R}. \exists i. \psi \ i = f$
by *auto*
then obtain psi **where** *psi*: *recfn 2 psi total psi eval psi* [*i, x*] = $\psi \ i \ x$ **for** $i \ x$
by *auto*
define d **where** $d = \text{Cn } 1 \ S \ [\text{Cn } 1 \ \text{psi} \ [\text{Id } 1 \ 0, \ \text{Id } 1 \ 0]]$
then have *recfn 1 d*
using *psi(1)* **by** *simp*
moreover have $d: \text{eval } d \ [x] \ \downarrow = \text{Suc} \ (\text{the} \ (\psi \ x \ x))$ **for** x
unfolding *d-def* **using** *num psi* **by** *simp*
ultimately have $(\lambda x. \text{eval } d \ [x]) \in \mathcal{R}$
using *R1I* **by** *blast*
then obtain i **where** $\psi \ i = (\lambda x. \text{eval } d \ [x])$
using *num(2)* **by** *auto*
then have $\psi \ i \ i = \text{eval } d \ [i]$ **by** *simp*
with d **have** $\psi \ i \ i \ \downarrow = \text{Suc} \ (\text{the} \ (\psi \ i \ i))$ **by** *simp*
then show *False*
using *option.sel*[*of Suc (the (psi i i))*] **by** *simp*

qed

A hypothesis space that contains a function for every prefix will come in handy. The following is a total numbering with this property.

definition *r-prenum* \equiv

$\text{Cn } 2 \ \text{r-iffless} \ [\text{Id } 2 \ 1, \ \text{Cn } 2 \ \text{r-length} \ [\text{Id } 2 \ 0], \ \text{Cn } 2 \ \text{r-nth} \ [\text{Id } 2 \ 0, \ \text{Id } 2 \ 1], \ \text{r-constrn } 1 \ 0]$

lemma *r-prenum-prim* [*simp*]: *prim-recfn 2 r-prenum*
unfolding *r-prenum-def* **by** *simp-all*

lemma *r-prenum* [*simp*]:
 $\text{eval } r\text{-prenum} \ [e, x] \ \downarrow = (\text{if } x < e\text{-length } e \text{ then } e\text{-nth } e \ x \ \text{else } 0)$
by (*simp add: r-prenum-def*)

definition *prenum* $::$ *partial2* **where**

$\text{prenum } e \ x \equiv \text{Some } (\text{if } x < \text{e-length } e \text{ then } e\text{-nth } e \ x \text{ else } 0)$

lemma *prenum-in-R2*: $\text{prenum} \in \mathcal{R}^2$
using *prenum-def Prim2I*[*OF r-prenum-prim, of prenum*] **by** *simp*

lemma *prenum* [*simp*]: $\text{prenum } e \ x \downarrow = (\text{if } x < \text{e-length } e \text{ then } e\text{-nth } e \ x \text{ else } 0)$
unfolding *prenum-def* ..

lemma *prenum-encode*:
 $\text{prenum } (\text{list-encode } vs) \ x \downarrow = (\text{if } x < \text{length } vs \text{ then } vs \ ! \ x \text{ else } 0)$
using *prenum-def* **by** (*cases* $x < \text{length } vs$) *simp-all*

Prepending a list of numbers to a function:

definition *prepend* :: $\text{nat list} \Rightarrow \text{partial1} \Rightarrow \text{partial1}$ (**infixr** $\langle \odot \rangle$ 64) **where**
 $vs \odot f \equiv \lambda x. \text{if } x < \text{length } vs \text{ then } \text{Some } (vs \ ! \ x) \text{ else } f \ (x - \text{length } vs)$

lemma *prepend* [*simp*]:
 $(vs \odot f) \ x = (\text{if } x < \text{length } vs \text{ then } \text{Some } (vs \ ! \ x) \text{ else } f \ (x - \text{length } vs))$
unfolding *prepend-def* ..

lemma *prepend-total*: $\text{total1 } f \Longrightarrow \text{total1 } (vs \odot f)$
unfolding *total1-def* **by** *simp*

lemma *prepend-at-less*:
assumes $n < \text{length } vs$
shows $(vs \odot f) \ n \downarrow = vs \ ! \ n$
using *assms* **by** *simp*

lemma *prepend-at-ge*:
assumes $n \geq \text{length } vs$
shows $(vs \odot f) \ n = f \ (n - \text{length } vs)$
using *assms* **by** *simp*

lemma *prefix-prepend-less*:
assumes $n < \text{length } vs$
shows $\text{prefix } (vs \odot f) \ n = \text{take } (\text{Suc } n) \ vs$
using *assms* *length-prefix* **by** (*intro nth-equalityI*) *simp-all*

lemma *prepend-eqI*:
assumes $\bigwedge x. x < \text{length } vs \Longrightarrow g \ x \downarrow = vs \ ! \ x$
and $\bigwedge x. g \ (\text{length } vs + x) = f \ x$
shows $g = vs \odot f$

proof
fix x
show $g \ x = (vs \odot f) \ x$
proof (*cases* $x < \text{length } vs$)
case *True*
then show *?thesis* **using** *assms* **by** *simp*
next
case *False*
then show *?thesis*
using *assms* *prepend* **by** (*metis add-diff-inverse-nat*)
qed
qed

fun *r-prepend* :: $\text{nat list} \Rightarrow \text{recf} \Rightarrow \text{recf}$ **where**

$r\text{-prepend [] } r = r$
 $| r\text{-prepend } (v \# vs) r =$
 $\quad Cn\ 1\ (r\text{-lifz } (r\text{-const } v) (Cn\ 1\ (r\text{-prepend } vs\ r) [r\text{-dec}])) [Id\ 1\ 0, Id\ 1\ 0]$

lemma *r-prepend-recfn*:
assumes *recfn 1 r*
shows *recfn 1 (r-prepend vs r)*
using *assms* **by** (*induction vs*) *simp-all*

lemma *r-prepend*:
assumes *recfn 1 r*
shows *eval (r-prepend vs r) [x] =*
(if x < length vs then Some (vs ! x) else eval r [x - length vs])
proof (*induction vs arbitrary: x*)
case *Nil*
then show *?case* **using** *assms* **by** *simp*
next
case (*Cons v vs*)
show *?case*
using *assms Cons* **by** (*cases x = 0*) (*auto simp add: r-prepend-recfn*)
qed

lemma *r-prepend-total*:
assumes *recfn 1 r* **and** *total r*
shows *eval (r-prepend vs r) [x] ↓ =*
(if x < length vs then vs ! x else the (eval r [x - length vs]))
proof (*induction vs arbitrary: x*)
case *Nil*
then show *?case* **using** *assms* **by** *simp*
next
case (*Cons v vs*)
show *?case*
using *assms Cons* **by** (*cases x = 0*) (*auto simp add: r-prepend-recfn*)
qed

lemma *prepend-in-P1*:
assumes $f \in \mathcal{P}$
shows $vs \odot f \in \mathcal{P}$
proof –
obtain *r* **where** $r: recfn\ 1\ r \wedge x. eval\ r\ [x] = f\ x$
using *assms* **by** *auto*
moreover have *recfn 1 (r-prepend vs r)*
using *r r-prepend-recfn* **by** *simp*
moreover have $eval\ (r-prepend\ vs\ r)\ [x] = (vs \odot f)\ x$ **for** *x*
using *r r-prepend* **by** *simp*
ultimately show *?thesis* **by** *blast*
qed

lemma *prepend-in-R1*:
assumes $f \in \mathcal{R}$
shows $vs \odot f \in \mathcal{R}$
proof –
obtain *r* **where** $r: recfn\ 1\ r\ total\ r \wedge x. eval\ r\ [x] = f\ x$
using *assms* **by** *auto*
then have *total1 f*
using *R1-imp-total1 [OF assms]* **by** *simp*

have *total* (*r-prepend vs r*)
using *r r-prepend-total r-prepend-recfn totalI1* [of *r-prepend vs r*] **by** *simp*
with *r* **have** *total* (*r-prepend vs r*) **by** *simp*
moreover **have** *recfn 1* (*r-prepend vs r*)
using *r r-prepend-recfn* **by** *simp*
moreover **have** *eval* (*r-prepend vs r*) [*x*] = (*vs* \odot *f*) *x* **for** *x*
using *r r-prepend* \langle *total1 f* \rangle *total1E* **by** *simp*
ultimately **show** *?thesis* **by** *auto*
qed

lemma *prepend-associative*: (*us* $\@$ *vs*) \odot *f* = *us* \odot *vs* \odot *f* (**is** *?lhs* = *?rhs*)

proof

fix *x*

consider

x < *length us*

| *x* \geq *length us* \wedge *x* < *length (us @ vs)*

| *x* \geq *length (us @ vs)*

by *linarith*

then **show** *?lhs x* = *?rhs x*

proof (*cases*)

case *1*

then **show** *?thesis*

by (*metis le-add1 length-append less-le-trans nth-append prepend-at-less*)

next

case *2*

then **show** *?thesis*

by (*smt add-diff-inverse-nat add-less-cancel-left length-append nth-append prepend*)

next

case *3*

then **show** *?thesis*

using *prepend-at-ge* **by** *auto*

qed

qed

abbreviation *constant-divergent* :: *partial1* ($\langle \uparrow^\infty \rangle$) **where**

$\uparrow^\infty \equiv \lambda\cdot$. *None*

abbreviation *constant-zero* :: *partial1* ($\langle 0^\infty \rangle$) **where**

$0^\infty \equiv \lambda\cdot$. *Some 0*

lemma *almost0-in-R1*: *vs* \odot $0^\infty \in \mathcal{R}$

using *RPred1-subseteq-R1 const0-in-RPred1 prepend-in-R1* **by** *auto*

The class U_0 of all total recursive functions that are almost everywhere zero will be used several times to construct (counter-)examples.

definition *U0* :: *partial1 set* ($\langle U_0 \rangle$) **where**

$U_0 \equiv \{vs \odot 0^\infty \mid vs. vs \in UNIV\}$

The class U_0 contains exactly the functions in the numbering *prenum*.

lemma *U0-altdef*: $U_0 = \{prenum\ e \mid e. e \in UNIV\}$ (**is** $U_0 = ?W$)

proof

show $U_0 \subseteq ?W$

proof

fix *f*

assume $f \in U_0$

with $U0\text{-def}$ **obtain** vs **where** $f = vs \odot 0^\infty$
by *auto*
then have $f = \text{prenum}$ (*list-encode vs*)
using *prenum-encode* **by** *auto*
then show $f \in ?W$ **by** *auto*
qed
show $?W \subseteq U_0$
unfolding $U0\text{-def}$ **by** *fastforce*
qed

lemma $U0\text{-in-NUM}$: $U_0 \in \text{NUM}$
using *prenum-in-R2 U0-altdef* **by** (*intro NUM-I[of prenum]; force*)

Every almost-zero function can be represented by $v0^\infty$ for a list v not ending in zero.

lemma *almost0-canonical*:

assumes $f = vs \odot 0^\infty$ **and** $f \neq 0^\infty$
obtains ws **where** $\text{length } ws > 0$ **and** $\text{last } ws \neq 0$ **and** $f = ws \odot 0^\infty$
proof –
let $?P = \lambda k. k < \text{length } vs \wedge vs ! k \neq 0$
from *assms* **have** $vs \neq []$
by *auto*
then have $ex: \exists k < \text{length } vs. vs ! k \neq 0$
using *assms* **by** *auto*
define m **where** $m = \text{Greatest } ?P$
moreover have $le: \forall y. ?P y \longrightarrow y \leq \text{length } vs$
by *simp*
ultimately have $?P m$
using ex *GreatestI-ex-nat[of ?P length vs]* **by** *simp*
have *not-gr*: $\neg ?P k$ **if** $k > m$ **for** k
using *Greatest-le-nat[of ?P - length vs]* $m\text{-def}$ ex *le not-less that* **by** *blast*
let $?ws = \text{take } (\text{Suc } m) \text{ } vs$
have $vs \odot 0^\infty = ?ws \odot 0^\infty$
proof
fix x
show $(vs \odot 0^\infty) x = (?ws \odot 0^\infty) x$
proof (*cases x < Suc m*)
case *True*
then show *?thesis* **using** $\langle ?P m \rangle$ **by** *simp*
next
case *False*
moreover from *this* **have** $(?ws \odot 0^\infty) x \downarrow = 0$
by *simp*
ultimately show *?thesis*
using *not-gr* **by** (*cases x < length vs*) *simp-all*
qed
qed
then have $f = ?ws \odot 0^\infty$
using *assms(1)* **by** *simp*
moreover have $\text{length } ?ws > 0$
by (*simp add: \langle vs \neq [] \rangle*)
moreover have $\text{last } ?ws \neq 0$
by (*simp add: \langle ?P m \rangle take-Suc-conv-app-nth*)
ultimately show *?thesis* **using** *that* **by** *blast*
qed

2.2 Types of inference

This section introduces all inference types that we are going to consider together with some of their simple properties. All these inference types share the following condition, which essentially says that everything must be computable:

abbreviation *environment* $:: \text{partial2} \Rightarrow (\text{partial1 set}) \Rightarrow \text{partial1} \Rightarrow \text{bool}$ **where**
environment $\psi U s \equiv \psi \in \mathcal{P}^2 \wedge U \subseteq \mathcal{R} \wedge s \in \mathcal{P} \wedge (\forall f \in U. \forall n. s (f \triangleright n) \downarrow)$

2.2.1 LIM: Learning in the limit

A strategy S learns a class U in the limit with respect to a hypothesis space $\psi \in \mathcal{P}^2$ if for all $f \in U$, the sequence $(S(f^n))_{n \in \mathbb{N}}$ converges to an i with $\psi_i = f$. Convergence for a sequence of natural numbers means that almost all elements are the same. We express this with the following notation.

abbreviation *Almost-All* $:: (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{bool}$ (**binder** $\langle \forall^\infty \rangle 10$) **where**
 $\forall^\infty n. P n \equiv \exists n_0. \forall n \geq n_0. P n$

definition *learn-lim* $:: \text{partial2} \Rightarrow (\text{partial1 set}) \Rightarrow \text{partial1} \Rightarrow \text{bool}$ **where**
learn-lim $\psi U s \equiv$
environment $\psi U s \wedge$
 $(\forall f \in U. \exists i. \psi i = f \wedge (\forall^\infty n. s (f \triangleright n) \downarrow = i))$

lemma *learn-limE*:

assumes *learn-lim* $\psi U s$
shows *environment* $\psi U s$
and $\bigwedge f. f \in U \implies \exists i. \psi i = f \wedge (\forall^\infty n. s (f \triangleright n) \downarrow = i)$
using *assms learn-lim-def* **by** *auto*

lemma *learn-limI*:

assumes *environment* $\psi U s$
and $\bigwedge f. f \in U \implies \exists i. \psi i = f \wedge (\forall^\infty n. s (f \triangleright n) \downarrow = i)$
shows *learn-lim* $\psi U s$
using *assms learn-lim-def* **by** *auto*

definition *LIM-wrt* $:: \text{partial2} \Rightarrow \text{partial1 set set}$ **where**
LIM-wrt $\psi \equiv \{U. \exists s. \text{learn-lim } \psi U s\}$

definition *Lim* $:: \text{partial1 set set}$ ($\langle \text{LIM} \rangle$) **where**
Lim $\equiv \{U. \exists \psi s. \text{learn-lim } \psi U s\}$

LIM is closed under the the subset relation.

lemma *learn-lim-closed-subseteq*:

assumes *learn-lim* $\psi U s$ **and** $V \subseteq U$
shows *learn-lim* $\psi V s$
using *assms learn-lim-def* **by** *auto*

corollary *LIM-closed-subseteq*:

assumes $U \in \text{LIM}$ **and** $V \subseteq U$
shows $V \in \text{LIM}$
using *assms learn-lim-closed-subseteq* **by** (*smt Lim-def mem-Collect-eq*)

Changing the hypothesis infinitely often precludes learning in the limit.

lemma *infinite-hyp-changes-not-Lim*:

assumes $f \in U$ **and** $\forall n. \exists m_1 > n. \exists m_2 > n. s (f \triangleright m_1) \neq s (f \triangleright m_2)$

shows $\neg \text{learn-lim } \psi \ U \ s$
using *assms learn-lim-def* **by** (*metis less-imp-le*)

lemma *always-hyp-change-not-Lim*:
assumes $\bigwedge x. s (f \triangleright (\text{Suc } x)) \neq s (f \triangleright x)$
shows $\neg \text{learn-lim } \psi \ \{f\} \ s$
using *assms learn-limE* **by** (*metis le-SucI order-refl singletonI*)

Guessing a wrong hypothesis infinitely often precludes learning in the limit.

lemma *infinite-hyp-wrong-not-Lim*:
assumes $f \in U$ **and** $\forall n. \exists m > n. \psi (\text{the } (s (f \triangleright m))) \neq f$
shows $\neg \text{learn-lim } \psi \ U \ s$
using *assms learn-limE* **by** (*metis less-imp-le option.sel*)

Converging to the same hypothesis on two functions precludes learning in the limit.

lemma *same-hyp-for-two-not-Lim*:
assumes $f_1 \in U$
and $f_2 \in U$
and $f_1 \neq f_2$
and $\forall n \geq n_1. s (f_1 \triangleright n) = h$
and $\forall n \geq n_2. s (f_2 \triangleright n) = h$
shows $\neg \text{learn-lim } \psi \ U \ s$
using *assms learn-limE* **by** (*metis le-cases option.sel*)

Every class that can be learned in the limit can be learned in the limit with respect to any Gödel numbering. We prove a generalization in which hypotheses may have to satisfy an extra condition, so we can re-use it for other inference types later.

lemma *learn-lim-extra-wrt-goedel*:
fixes *extra* :: (*partial1 set*) \Rightarrow *partial1* \Rightarrow *nat* \Rightarrow *partial1* \Rightarrow *bool*
assumes *goedel-numbering* χ
and *learn-lim* $\psi \ U \ s$
and $\bigwedge f n. f \in U \Longrightarrow \text{extra } U \ f \ n \ (\psi (\text{the } (s (f \triangleright n))))$
shows $\exists t. \text{learn-lim } \chi \ U \ t \wedge (\forall f \in U. \forall n. \text{extra } U \ f \ n \ (\chi (\text{the } (t (f \triangleright n))))))$

proof –

have *env*: *environment* $\psi \ U \ s$
and *lim*: *learn-lim* $\psi \ U \ s$
and *extra*: $\forall f \in U. \forall n. \text{extra } U \ f \ n \ (\psi (\text{the } (s (f \triangleright n))))$
using *assms learn-limE* **by** *auto*
obtain *c* **where** $c: c \in \mathcal{R} \ \forall i. \psi \ i = \chi (\text{the } (c \ i))$
using *env goedel-numberingE*[*OF assms(1)*], *of* ψ **by** *auto*
define *t* **where** $t \equiv$
 $(\lambda x. \text{if } s \ x \ \downarrow \wedge \ c \ (\text{the } (s \ x)) \ \downarrow \ \text{then } \text{Some } (\text{the } (c \ (\text{the } (s \ x)))) \ \text{else } \text{None})$

have $t \in \mathcal{P}$

unfolding *t-def* **using** *env c concat-P1-P1*[*of c s*] **by** *auto*
have $t \ x = (\text{if } s \ x \ \downarrow \ \text{then } \text{Some } (\text{the } (c \ (\text{the } (s \ x)))) \ \text{else } \text{None})$ **for** x
using *t-def c(1) R1-imp-total1* **by** *auto*
then have $t: t (f \triangleright n) \downarrow = \text{the } (c \ (\text{the } (s (f \triangleright n))))$ **if** $f \in U$ **for** $f \ n$
using *lim learn-limE that* **by** *simp*

have *learn-lim* $\chi \ U \ t$

proof (*rule learn-limI*)

show *environment* $\chi \ U \ t$
using *t* **by** (*simp add: t* $\in \mathcal{P}$) *env goedel-numbering-P2*[*OF assms(1)*]

show $\exists i. \chi \ i = f \wedge (\forall^\infty n. t (f \triangleright n) \downarrow = i)$ **if** $f \in U$ **for** f

proof –

from *lim learn-limE(2)* **obtain** $i \ n_0$ **where**

$i: \psi \ i = f \wedge (\forall n \geq n_0. s \ (f \triangleright n) \downarrow = i)$
using $\langle f \in U \rangle$ **by** *blast*
let $?j = \text{the } (c \ i)$
have $\chi \ ?j = f$
using $c(2) \ i$ **by** *simp*
moreover have $t \ (f \triangleright n) \downarrow = ?j$ **if** $n \geq n_0$ **for** n
by (*simp add: $\langle f \in U \rangle \ i \ t$ that*)
ultimately show *?thesis* **by** *auto*
qed
qed
moreover have *extra* $U \ f \ n \ (\chi \ (\text{the } (t \ (f \triangleright n))))$ **if** $f \in U$ **for** $f \ n$
proof –
from t **have** $\text{the } (t \ (f \triangleright n)) = \text{the } (c \ (\text{the } (s \ (f \triangleright n))))$
by (*simp add: that*)
then have $\chi \ (\text{the } (t \ (f \triangleright n))) = \psi \ (\text{the } (s \ (f \triangleright n)))$
using $c(2)$ **by** *simp*
with *extra* **show** *?thesis* **using** *that* **by** *simp*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *learn-lim-wrt-goedel*:
assumes *goedel-numbering* χ **and** *learn-lim* $\psi \ U \ s$
shows $\exists t. \text{learn-lim } \chi \ U \ t$
using *assms learn-lim-extra-wrt-goedel* [**where** $?extra = \lambda U \ f \ n \ h. \ \text{True}$]
by *simp*

lemma *LIM-wrt-phi-eq-Lim*: $LIM\text{-wrt } \varphi = LIM$
using *LIM-wrt-def Lim-def learn-lim-wrt-goedel* [*OF goedel-numbering-phi*]
by *blast*

2.2.2 BC: Behaviorally correct learning in the limit

Behaviorally correct learning in the limit relaxes LIM by requiring that the strategy almost always output an index for the target function, but not necessarily the same index. In other words convergence of $(S(f^n))_{n \in \mathbb{N}}$ is replaced by convergence of $(\psi_{S(f^n)})_{n \in \mathbb{N}}$.

definition *learn-bc* :: *partial2* \Rightarrow (*partial1 set*) \Rightarrow *partial1* \Rightarrow *bool* **where**
 $\text{learn-bc } \psi \ U \ s \equiv$
 $\text{environment } \psi \ U \ s \wedge$
 $(\forall f \in U. \forall^\infty n. \psi \ (\text{the } (s \ (f \triangleright n))) = f)$

lemma *learn-bcE*:
assumes *learn-bc* $\psi \ U \ s$
shows *environment* $\psi \ U \ s$
and $\bigwedge f. f \in U \Longrightarrow \forall^\infty n. \psi \ (\text{the } (s \ (f \triangleright n))) = f$
using *assms learn-bc-def* **by** *auto*

lemma *learn-bcI*:
assumes *environment* $\psi \ U \ s$
and $\bigwedge f. f \in U \Longrightarrow \forall^\infty n. \psi \ (\text{the } (s \ (f \triangleright n))) = f$
shows *learn-bc* $\psi \ U \ s$
using *assms learn-bc-def* **by** *auto*

definition *BC-wrt* :: *partial2* \Rightarrow *partial1 set set* **where**
 $BC\text{-wrt } \psi \equiv \{U. \exists s. \text{learn-bc } \psi \ U \ s\}$

definition *BC* :: *partial1 set set* **where**

$BC \equiv \{U. \exists \psi s. \text{learn-bc } \psi \ U \ s\}$

BC is a superset of LIM and closed under the subset relation.

lemma *learn-lim-imp-BC*: *learn-lim* $\psi \ U \ s \implies \text{learn-bc } \psi \ U \ s$

using *learn-limE learn-bcI*[of $\psi \ U \ s$] **by** *fastforce*

lemma *Lim-subseteq-BC*: $LIM \subseteq BC$

using *learn-lim-imp-BC Lim-def BC-def* **by** *blast*

lemma *learn-bc-closed-subseteq*:

assumes *learn-bc* $\psi \ U \ s$ **and** $V \subseteq U$

shows *learn-bc* $\psi \ V \ s$

using *assms learn-bc-def* **by** *auto*

corollary *BC-closed-subseteq*:

assumes $U \in BC$ **and** $V \subseteq U$

shows $V \in BC$

using *assms* **by** (*smt BC-def learn-bc-closed-subseteq mem-Collect-eq*)

Just like with LIM, guessing a wrong hypothesis infinitely often precludes BC-style learning.

lemma *infinite-hyp-wrong-not-BC*:

assumes $f \in U$ **and** $\forall n. \exists m > n. \psi (\text{the } (s \ (f \triangleright m))) \neq f$

shows $\neg \text{learn-bc } \psi \ U \ s$

proof

assume *learn-bc* $\psi \ U \ s$

then obtain n_0 **where** $\forall n \geq n_0. \psi (\text{the } (s \ (f \triangleright n))) = f$

using *learn-bcE assms(1)* **by** *metis*

with *assms(2)* **show** *False* **using** *less-imp-le* **by** *blast*

qed

The proof that Gödel numberings suffice as hypothesis spaces for BC is similar to the one for *learn-lim-extra-wrt-goedel*. We do not need the *extra* part for BC, but we get it for free.

lemma *learn-bc-extra-wrt-goedel*:

fixes *extra* :: (*partial1 set*) \Rightarrow *partial1* \Rightarrow *nat* \Rightarrow *partial1* \Rightarrow *bool*

assumes *goedel-numbering* χ

and *learn-bc* $\psi \ U \ s$

and $\bigwedge f n. f \in U \implies \text{extra } U \ f \ n \ (\psi (\text{the } (s \ (f \triangleright n))))$

shows $\exists t. \text{learn-bc } \chi \ U \ t \wedge (\forall f \in U. \forall n. \text{extra } U \ f \ n \ (\chi (\text{the } (t \ (f \triangleright n))))$

proof –

have *env*: *environment* $\psi \ U \ s$

and *lim*: *learn-bc* $\psi \ U \ s$

and *extra*: $\forall f \in U. \forall n. \text{extra } U \ f \ n \ (\psi (\text{the } (s \ (f \triangleright n))))$

using *assms learn-bc-def* **by** *auto*

obtain c **where** $c: c \in \mathcal{R} \ \forall i. \psi \ i = \chi (\text{the } (c \ i))$

using *env goedel-numberingE*[*OF assms(1)*, of ψ] **by** *auto*

define t **where**

$t = (\lambda x. \text{if } s \ x \ \downarrow \wedge c \ (\text{the } (s \ x)) \ \downarrow \text{ then } \text{Some } (\text{the } (c \ (\text{the } (s \ x)))) \text{ else } \text{None})$

have $t \in \mathcal{P}$

unfolding *t-def* **using** *env c concat-P1-P1*[of $c \ s$] **by** *auto*

have $t \ x = (\text{if } s \ x \ \downarrow \text{ then } \text{Some } (\text{the } (c \ (\text{the } (s \ x)))) \text{ else } \text{None})$ **for** x

using *t-def c(1) R1-imp-total1* **by** *auto*

then have $t: t (f \triangleright n) \Downarrow = \text{the} (c (the (s (f \triangleright n))))$ **if** $f \in U$ **for** $f n$
using $\text{lim learn-bcE}(1)$ **that by** simp
have $\text{learn-bc } \chi U t$
proof (rule learn-bcI)
show $\text{environment } \chi U t$
using t **by** ($\text{simp add: } \langle t \in \mathcal{P} \rangle \text{ env goedel-numbering-P2}[OF \text{ assms}(1)]$)
show $\forall^\infty n. \chi (the (t (f \triangleright n))) = f$ **if** $f \in U$ **for** f
proof –
obtain n_0 **where** $\forall n \geq n_0. \psi (the (s (f \triangleright n))) = f$
using $\text{lim learn-bcE}(2)$ $\langle f \in U \rangle$ **by** blast
then show $?thesis$ **using** $that t c(2)$ **by** auto
qed
qed
moreover have $\text{extra } U f n (\chi (the (t (f \triangleright n))))$ **if** $f \in U$ **for** $f n$
proof –
from t **have** $the (t (f \triangleright n)) = the (c (the (s (f \triangleright n))))$
by (simp add: that)
then have $\chi (the (t (f \triangleright n))) = \psi (the (s (f \triangleright n)))$
using $c(2)$ **by** simp
with $\text{extra show } ?thesis$ **using** $that$ **by** simp
qed
ultimately show $?thesis$ **by** auto
qed

corollary $\text{learn-bc-wrt-goedel}$:

assumes $\text{goedel-numbering } \chi$ **and** $\text{learn-bc } \psi U s$
shows $\exists t. \text{learn-bc } \chi U t$
using $\text{assms learn-bc-extra-wrt-goedel}$ **where** $?extra = \lambda \dots \text{True}$ **by** simp

corollary $BC\text{-wrt-phi-eq-BC}$: $BC\text{-wrt } \varphi = BC$

using $\text{learn-bc-wrt-goedel goedel-numbering-phi BC-def BC-wrt-def}$ **by** blast

2.2.3 CONS: Learning in the limit with consistent hypotheses

A hypothesis is *consistent* if it matches all values in the prefix given to the strategy. Consistent learning in the limit requires the strategy to output only consistent hypotheses for prefixes from the class.

definition $\text{learn-cons} :: \text{partial2} \Rightarrow (\text{partial1 set}) \Rightarrow \text{partial1} \Rightarrow \text{bool}$ **where**

$\text{learn-cons } \psi U s \equiv$
 $\text{learn-lim } \psi U s \wedge$
 $(\forall f \in U. \forall n. \forall k \leq n. \psi (the (s (f \triangleright n))) k = f k)$

definition $CONS\text{-wrt} :: \text{partial2} \Rightarrow \text{partial1 set set}$ **where**

$CONS\text{-wrt } \psi \equiv \{U. \exists s. \text{learn-cons } \psi U s\}$

definition $CONS :: \text{partial1 set set}$ **where**

$CONS \equiv \{U. \exists \psi s. \text{learn-cons } \psi U s\}$

lemma $CONS\text{-subsetq-Lim}$: $CONS \subseteq LIM$

using $CONS\text{-def Lim-def learn-cons-def}$ **by** blast

lemma learn-consI :

assumes $\text{environment } \psi U s$
and $\bigwedge f. f \in U \implies \exists i. \psi i = f \wedge (\forall^\infty n. s (f \triangleright n) \Downarrow = i)$
and $\bigwedge f n. f \in U \implies \forall k \leq n. \psi (the (s (f \triangleright n))) k = f k$

shows *learn-cons* ψ U s
using *assms learn-lim-def learn-cons-def* **by** *simp*

If a consistent strategy converges, it automatically converges to a correct hypothesis.
Thus we can remove ψ $i = f$ from the second assumption in the previous lemma.

lemma *learn-consI2*:

assumes *environment* ψ U s
and $\bigwedge f. f \in U \implies \exists i. \forall^\infty n. s (f \triangleright n) \downarrow = i$
and $\bigwedge f n. f \in U \implies \forall k \leq n. \psi (the (s (f \triangleright n))) k = f k$
shows *learn-cons* ψ U s
proof (*rule learn-consI*)
show *environment* ψ U s
and *cons*: $\bigwedge f n. f \in U \implies \forall k \leq n. \psi (the (s (f \triangleright n))) k = f k$
using *assms* **by** *simp-all*
show $\exists i. \psi$ $i = f \wedge (\forall^\infty n. s (f \triangleright n) \downarrow = i)$ **if** $f \in U$ **for** f
proof –
from *that assms(2)* **obtain** i n_0 **where** i - n_0 : $\forall n \geq n_0. s (f \triangleright n) \downarrow = i$
by *blast*
have ψ i $x = f$ x **for** x
proof (*cases* $x \leq n_0$)
case *True*
then show *?thesis*
using i - n_0 *cons* **that** **by** *fastforce*
next
case *False*
moreover have $\forall k \leq x. \psi (the (s (f \triangleright x))) k = f k$
using *cons* **that** **by** *simp*
ultimately show *?thesis* **using** i - n_0 **by** *simp*
qed
with i - n_0 **show** *?thesis* **by** *auto*
qed
qed

lemma *learn-consE*:

assumes *learn-cons* ψ U s
shows *environment* ψ U s
and $\bigwedge f. f \in U \implies \exists i n_0. \psi$ $i = f \wedge (\forall n \geq n_0. s (f \triangleright n) \downarrow = i)$
and $\bigwedge f n. f \in U \implies \forall k \leq n. \psi (the (s (f \triangleright n))) k = f k$
using *assms learn-cons-def learn-lim-def* **by** *auto*

lemma *learn-cons-wrt-goedel*:

assumes *goedel-numbering* χ **and** *learn-cons* ψ U s
shows $\exists t. learn-cons$ χ U t
using *learn-cons-def assms*
learn-lim-extra-wrt-goedel[**where** $?extra = \lambda U f n h. \forall k \leq n. h k = f k$]
by *auto*

lemma *CONS-wrt-phi-eq-CONS*: *CONS-wrt* $\varphi = CONS$

using *CONS-wrt-def CONS-def learn-cons-wrt-goedel goedel-numbering-phi*
by *blast*

lemma *learn-cons-closed-subseteq*:

assumes *learn-cons* ψ U s **and** $V \subseteq U$
shows *learn-cons* ψ V s
using *assms learn-cons-def learn-lim-closed-subseteq* **by** *auto*

lemma *CONS-closed-subseteq*:
assumes $U \in \text{CONS}$ **and** $V \subseteq U$
shows $V \in \text{CONS}$
using *assms learn-cons-closed-subseteq* **by** (*smt CONS-def mem-Collect-eq*)

A consistent strategy cannot output the same hypothesis for two different prefixes from the class to be learned.

lemma *same-hyp-different-init-not-cons*:
assumes $f \in U$
and $g \in U$
and $f \triangleright n \neq g \triangleright n$
and $s(f \triangleright n) = s(g \triangleright n)$
shows $\neg \text{learn-cons } \varphi U s$
unfolding *learn-cons-def* **by** (*auto,metis assms init-eqI*)

2.2.4 TOTAL: Learning in the limit with total hypotheses

Total learning in the limit requires the strategy to hypothesize only total functions for prefixes from the class.

definition *learn-total* :: *partial2* \Rightarrow (*partial1 set*) \Rightarrow *partial1* \Rightarrow *bool* **where**
learn-total $\psi U s \equiv$
learn-lim $\psi U s \wedge$
 $(\forall f \in U. \forall n. \psi(\text{the}(s(f \triangleright n)))) \in \mathcal{R}$

definition *TOTAL-wrt* :: *partial2* \Rightarrow *partial1 set set* **where**
TOTAL-wrt $\psi \equiv \{U. \exists s. \text{learn-total } \psi U s\}$

definition *TOTAL* :: *partial1 set set* **where**
TOTAL $\equiv \{U. \exists \psi s. \text{learn-total } \psi U s\}$

lemma *TOTAL-subseteq-LIM*: $TOTAL \subseteq LIM$
unfolding *TOTAL-def Lim-def* **using** *learn-total-def* **by** *auto*

lemma *learn-totalI*:
assumes *environment* $\psi U s$
and $\bigwedge f. f \in U \implies \exists i. \psi i = f \wedge (\forall^\infty n. s(f \triangleright n) \downarrow = i)$
and $\bigwedge f n. f \in U \implies \psi(\text{the}(s(f \triangleright n))) \in \mathcal{R}$
shows *learn-total* $\psi U s$
using *assms learn-lim-def learn-total-def* **by** *auto*

lemma *learn-totalE*:
assumes *learn-total* $\psi U s$
shows *environment* $\psi U s$
and $\bigwedge f. f \in U \implies \exists i n_0. \psi i = f \wedge (\forall n \geq n_0. s(f \triangleright n) \downarrow = i)$
and $\bigwedge f n. f \in U \implies \psi(\text{the}(s(f \triangleright n))) \in \mathcal{R}$
using *assms learn-lim-def learn-total-def* **by** *auto*

lemma *learn-total-wrt-goedel*:
assumes *goedel-numbering* χ **and** *learn-total* $\psi U s$
shows $\exists t. \text{learn-total } \chi U t$
using *learn-total-def assms learn-lim-extra-wrt-goedel* [**where** *?extra* $= \lambda U f n h. h \in \mathcal{R}$]
by *auto*

lemma *TOTAL-wrt-phi-eq-TOTAL*: $TOTAL\text{-wrt } \varphi = TOTAL$
using *TOTAL-wrt-def TOTAL-def learn-total-wrt-goedel goedel-numbering-phi*

by *blast*

lemma *learn-total-closed-subseteq*:

assumes *learn-total* ψ U s **and** $V \subseteq U$

shows *learn-total* ψ V s

using *assms learn-total-def learn-lim-closed-subseteq* **by** *auto*

lemma *TOTAL-closed-subseteq*:

assumes $U \in \text{TOTAL}$ **and** $V \subseteq U$

shows $V \in \text{TOTAL}$

using *assms learn-total-closed-subseteq* **by** (*smt TOTAL-def mem-Collect-eq*)

2.2.5 CP: Learning in the limit with class-preserving hypotheses

Class-preserving learning in the limit requires all hypotheses for prefixes from the class to be functions from the class.

definition *learn-cp* :: *partial2* \Rightarrow (*partial1 set*) \Rightarrow *partial1* \Rightarrow *bool* **where**

learn-cp ψ U s \equiv

learn-lim ψ U s \wedge

$(\forall f \in U. \forall n. \psi (\text{the } (s (f \triangleright n))) \in U)$

definition *CP-wrt* :: *partial2* \Rightarrow *partial1 set set* **where**

CP-wrt ψ \equiv $\{U. \exists s. \text{learn-cp } \psi U s\}$

definition *CP* :: *partial1 set set* **where**

CP \equiv $\{U. \exists \psi s. \text{learn-cp } \psi U s\}$

lemma *learn-cp-wrt-goedel*:

assumes *goedel-numbering* χ **and** *learn-cp* ψ U s

shows $\exists t. \text{learn-cp } \chi U t$

using *learn-cp-def assms learn-lim-extra-wrt-goedel* [**where** $?extra = \lambda U f n h. h \in U$]

by *auto*

corollary *CP-wrt-phi*: $CP = CP\text{-wrt } \varphi$

using *learn-cp-wrt-goedel* [*OF goedel-numbering-phi*]

by (*smt CP-def CP-wrt-def Collect-cong*)

lemma *learn-cpI*:

assumes *environment* ψ U s

and $\bigwedge f. f \in U \Longrightarrow \exists i. \psi i = f \wedge (\forall n \geq n_0. s (f \triangleright n) \downarrow = i)$

and $\bigwedge f n. f \in U \Longrightarrow \psi (\text{the } (s (f \triangleright n))) \in U$

shows *learn-cp* ψ U s

using *assms learn-cp-def learn-lim-def* **by** *auto*

lemma *learn-cpE*:

assumes *learn-cp* ψ U s

shows *environment* ψ U s

and $\bigwedge f. f \in U \Longrightarrow \exists i n_0. \psi i = f \wedge (\forall n \geq n_0. s (f \triangleright n) \downarrow = i)$

and $\bigwedge f n. f \in U \Longrightarrow \psi (\text{the } (s (f \triangleright n))) \in U$

using *assms learn-lim-def learn-cp-def* **by** *auto*

Since classes contain only total functions, a CP strategy is also a TOTAL strategy.

lemma *learn-cp-imp-total*: *learn-cp* ψ U s \Longrightarrow *learn-total* ψ U s

using *learn-cp-def learn-total-def learn-lim-def* **by** *auto*

lemma *CP-subseteq-TOTAL*: $CP \subseteq TOTAL$
using *learn-cp-imp-total CP-def TOTAL-def* **by** *blast*

2.2.6 FIN: Finite learning

In general it is undecidable whether a LIM strategy has reached its final hypothesis. By contrast, in finite learning (also called “one-shot learning”) the strategy signals when it is ready to output a hypothesis. Up until then it outputs a “don’t know yet” value. This value is represented by zero and the actual hypothesis i by $i + 1$.

definition *learn-fin* :: *partial2* \Rightarrow *partial1 set* \Rightarrow *partial1* \Rightarrow *bool* **where**

learn-fin ψ U s \equiv
environment ψ U s \wedge
 $(\forall f \in U. \exists i n_0. \psi i = f \wedge (\forall n < n_0. s (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. s (f \triangleright n) \downarrow = Suc i))$

definition *FIN-wrt* :: *partial2* \Rightarrow *partial1 set set* **where**

FIN-wrt $\psi \equiv \{U. \exists s. \text{learn-fin } \psi U s\}$

definition *FIN* :: *partial1 set set* **where**

FIN $\equiv \{U. \exists \psi s. \text{learn-fin } \psi U s\}$

lemma *learn-finI*:

assumes *environment* ψ U s

and $\bigwedge f. f \in U \implies$

$\exists i n_0. \psi i = f \wedge (\forall n < n_0. s (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. s (f \triangleright n) \downarrow = Suc i)$

shows *learn-fin* ψ U s

using *assms learn-fin-def* **by** *auto*

lemma *learn-finE*:

assumes *learn-fin* ψ U s

shows *environment* ψ U s

and $\bigwedge f. f \in U \implies$

$\exists i n_0. \psi i = f \wedge (\forall n < n_0. s (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. s (f \triangleright n) \downarrow = Suc i)$

using *assms learn-fin-def* **by** *auto*

lemma *learn-fin-closed-subseteq*:

assumes *learn-fin* ψ U s **and** $V \subseteq U$

shows *learn-fin* ψ V s

using *assms learn-fin-def* **by** *auto*

lemma *learn-fin-wrt-goedel*:

assumes *goedel-numbering* χ **and** *learn-fin* ψ U s

shows $\exists t. \text{learn-fin } \chi U t$

proof –

have *env*: *environment* ψ U s

and *fin*: $\bigwedge f. f \in U \implies$

$\exists i n_0. \psi i = f \wedge (\forall n < n_0. s (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. s (f \triangleright n) \downarrow = Suc i)$

using *assms(2) learn-finE* **by** *auto*

obtain c **where** $c: c \in \mathcal{R} \forall i. \psi i = \chi$ (*the* $(c i)$)

using *env goedel-numberingE[OF assms(1), of ψ]* **by** *auto*

define t **where** $t \equiv$

$\lambda x. \text{if } s x \uparrow \text{ then None}$

$\text{else if } s x = \text{Some } 0 \text{ then Some } 0$

$\text{else Some } (Suc (\text{the } (c (\text{the } (s x) - 1))))$

have $t \in \mathcal{P}$

proof –

```

from  $c$  obtain  $rc$  where  $rc$ :
   $recfn$  1  $rc$ 
   $total$   $rc$ 
   $\forall x. c\ x = eval\ rc\ [x]$ 
  by  $auto$ 
from  $env$  obtain  $rs$  where  $rs$ :  $recfn$  1  $rs\ \forall x. s\ x = eval\ rs\ [x]$ 
  by  $auto$ 
then have  $eval\ rs\ [f \triangleright n] \downarrow$  if  $f \in U$  for  $f\ n$ 
  using  $env\ that$  by  $simp$ 
define  $rt$  where  $rt = Cn\ 1\ r-ifz\ [rs, Z, Cn\ 1\ S\ [Cn\ 1\ rc\ [Cn\ 1\ r-dec\ [rs]]]]$ 
then have  $recfn\ 1\ rt$ 
  using  $rc(1)\ rs(1)$  by  $simp$ 
have  $eval\ rt\ [x] \uparrow$  if  $eval\ rs\ [x] \uparrow$  for  $x$ 
  using  $rc(1)\ rs(1)\ rt-def$  that by  $auto$ 
moreover have  $eval\ rt\ [x] \downarrow = 0$  if  $eval\ rs\ [x] \downarrow = 0$  for  $x$ 
  using  $rt-def\ that\ rc(1,2)\ rs(1)$  by  $simp$ 
moreover have  $eval\ rt\ [x] \downarrow = Suc\ (the\ (c\ (the\ (s\ x) - 1)))$  if  $eval\ rs\ [x] \downarrow \neq 0$  for  $x$ 
  using  $rt-def\ that\ rc\ rs$  by  $auto$ 
ultimately have  $eval\ rt\ [x] = t\ x$  for  $x$ 
  by ( $simp\ add: rs(2)\ t-def$ )
with  $\langle recfn\ 1\ rt \rangle$  show  $?thesis$  by  $auto$ 
qed
have  $t: t\ (f \triangleright n) \downarrow =$ 
  ( $if\ s\ (f \triangleright n) = Some\ 0\ then\ 0\ else\ Suc\ (the\ (c\ (the\ (s\ (f \triangleright n)) - 1)))$ )
if  $f \in U$  for  $f\ n$ 
  using  $that\ env$  by ( $simp\ add: t-def$ )
have  $learn-fin\ \chi\ U\ t$ 
proof ( $rule\ learn-finI$ )
  show  $environment\ \chi\ U\ t$ 
  using  $t$  by ( $simp\ add: \langle t \in \mathcal{P} \rangle\ env\ goedel-numbering-P2[OF\ assms(1)]$ )
  show  $\exists i\ n_0. \chi\ i = f \wedge (\forall n < n_0. t\ (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. t\ (f \triangleright n) \downarrow = Suc\ i)$ 
  if  $f \in U$  for  $f$ 
  proof -
    from  $fin$  obtain  $i\ n_0$  where
       $i: \psi\ i = f \wedge (\forall n < n_0. s\ (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. s\ (f \triangleright n) \downarrow = Suc\ i)$ 
      using  $\langle f \in U \rangle$  by  $blast$ 
    let  $?j = the\ (c\ i)$ 
    have  $\chi\ ?j = f$ 
      using  $c(2)\ i$  by  $simp$ 
    moreover have  $\forall n < n_0. t\ (f \triangleright n) \downarrow = 0$ 
      using  $t[OF\ that]\ i$  by  $simp$ 
    moreover have  $t\ (f \triangleright n) \downarrow = Suc\ ?j$  if  $n \geq n_0$  for  $n$ 
      using  $that\ i\ t[OF\ \langle f \in U \rangle]$  by  $simp$ 
    ultimately show  $?thesis$  by  $auto$ 
  qed
qed
then show  $?thesis$  by  $auto$ 
qed
end

```

2.3 FIN is a proper subset of CP

```

theory  $CP-FIN-NUM$ 
imports  $Inductive-Inference-Basics$ 

```

begin

Let S be a FIN strategy for a non-empty class U . Let T be a strategy that hypothesizes an arbitrary function from U while S outputs “don’t know” and the hypothesis of S otherwise. Then T is a CP strategy for U .

lemma *nonempty-FIN-wrt-impl-CP*:

assumes $U \neq \{\}$ **and** $U \in \text{FIN-wrt } \psi$
shows $U \in \text{CP-wrt } \psi$

proof –

obtain s **where** *learn-fin* ψ U s

using *assms(2)* *FIN-wrt-def* **by** *auto*

then have *env: environment* ψ U s **and**

fin: $\bigwedge f. f \in U \implies$

$\exists i n_0. \psi i = f \wedge (\forall n < n_0. s (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. s (f \triangleright n) \downarrow = \text{Suc } i)$

using *learn-finE* **by** *auto*

from *assms(1)* **obtain** f_0 **where** $f_0 \in U$

by *auto*

with *fin* **obtain** i_0 **where** $\psi i_0 = f_0$

by *blast*

define t **where** $t x \equiv$

(if $s x \uparrow$ *then* *None* *else if* $s x \downarrow = 0$ *then* *Some* i_0 *else* *Some* *(the* $(s x) - 1$ *)**)*

for x

have $t \in \mathcal{P}$

proof –

from *env* **obtain** rs **where** $rs: \text{recfn } 1 rs \bigwedge x. \text{eval } rs [x] = s x$

by *auto*

define rt **where** $rt = \text{Cn } 1 \text{ r-ifz } [rs, \text{r-const } i_0, \text{Cn } 1 \text{ r-dec } [rs]]$

then have *recfn* $1 rt$

using *rs(1)* **by** *simp*

then have *eval* $rt [x] \downarrow = (\text{if } s x \downarrow = 0 \text{ then } i_0 \text{ else } (\text{the } (s x)) - 1)$ **if** $s x \downarrow$ **for** x

using *rs rt-def* **that** **by** *auto*

moreover have *eval* $rt [x] \uparrow$ **if** *eval* $rs [x] \uparrow$ **for** x

using *rs rt-def* **that** **by** *simp*

ultimately have *eval* $rt [x] = t x$ **for** x

using *rs(2)* *t-def* **by** *simp*

with $\langle \text{recfn } 1 rt \rangle$ **show** *?thesis* **by** *auto*

qed

have *learn-cp* ψ U t

proof (*rule learn-cpI*)

show *environment* ψ U t

using *env t-def* $\langle t \in \mathcal{P} \rangle$ **by** *simp*

show $\exists i. \psi i = f \wedge (\forall n. t (f \triangleright n) \downarrow = i)$ **if** $f \in U$ **for** f

proof –

from *that fin* **obtain** $i n_0$ **where**

$i: \psi i = f \forall n < n_0. s (f \triangleright n) \downarrow = 0 \forall n \geq n_0. s (f \triangleright n) \downarrow = \text{Suc } i$

by *blast*

moreover have $\forall n \geq n_0. t (f \triangleright n) \downarrow = i$

using *that t-def* $i(3)$ **by** *simp*

ultimately show *?thesis* **by** *auto*

qed

show $\psi (\text{the } (t (f \triangleright n))) \in U$ **if** $f \in U$ **for** $f n$

using $\langle \psi i_0 = f_0 \rangle \langle f_0 \in U \rangle$ *t-def fin env* **that**

by (*metis* (*no-types*, *lifting*) *diff-Suc-1 not-less option.sel*)

qed

then show *?thesis* **using** *CP-wrt-def env* **by** *auto*

qed

lemma *FIN-wrt-impl-CP*:
assumes $U \in \text{FIN-wrt } \psi$
shows $U \in \text{CP-wrt } \psi$
proof (*cases* $U = \{\}$)
case *True*
then have $\psi \in \mathcal{P}^2 \implies U \in \text{CP-wrt } \psi$
using *CP-wrt-def learn-cpI*[of $\psi \{\}$ $\lambda x. \text{Some } 0$] *const-in-Prim1* **by** *auto*
moreover have $\psi \in \mathcal{P}^2$
using *assms FIN-wrt-def learn-finE* **by** *auto*
ultimately show $U \in \text{CP-wrt } \psi$ **by** *simp*
next
case *False*
with *nonempty-FIN-wrt-impl-CP assms* **show** *?thesis*
by *simp*
qed

corollary *FIN-subseteq-CP*: $\text{FIN} \subseteq \text{CP}$

proof
fix U
assume $U \in \text{FIN}$
then have $\exists \psi. U \in \text{FIN-wrt } \psi$
using *FIN-def FIN-wrt-def* **by** *auto*
then have $\exists \psi. U \in \text{CP-wrt } \psi$
using *FIN-wrt-impl-CP* **by** *auto*
then show $U \in \text{CP}$
by (*simp add: CP-def CP-wrt-def*)
qed

In order to show the *proper* inclusion, we show $U_0 \in \text{CP} - \text{FIN}$. A CP strategy for U_0 simply hypothesizes the function in U_0 with the longest prefix of f^n not ending in zero. For that we define a function computing the index of the rightmost non-zero value in a list, returning the length of the list if there is no such value.

definition *findr* :: *partial1* **where**
findr e \equiv
if $\exists i < e\text{-length } e. e\text{-nth } e \ i \neq 0$
then *Some (GREATEST i. i < e-length e \wedge e-nth e i \neq 0)*
else *Some (e-length e)*

lemma *findr-total*: *findr e* \downarrow
unfolding *findr-def* **by** *simp*

lemma *findr-ex*:
assumes $\exists i < e\text{-length } e. e\text{-nth } e \ i \neq 0$
shows *the (findr e) < e-length e*
and $e\text{-nth } e \ (\text{the } (\text{findr } e)) \neq 0$
and $\forall i. \text{the } (\text{findr } e) < i \wedge i < e\text{-length } e \longrightarrow e\text{-nth } e \ i = 0$
proof –
let $?P = \lambda i. i < e\text{-length } e \wedge e\text{-nth } e \ i \neq 0$
from *assms* **have** $\exists i. ?P \ i$ **by** *simp*
then have $?P \ (\text{Greatest } ?P)$
using *GreatestI-ex-nat*[of $?P \ e\text{-length } e$] **by** *fastforce*
moreover have $*$: $\text{findr } e = \text{Some } (\text{Greatest } ?P)$
using *assms findr-def* **by** *simp*
ultimately show *the (findr e) < e-length e* **and** $e\text{-nth } e \ (\text{the } (\text{findr } e)) \neq 0$

by *fastforce+*
show $\forall i. \text{the } (\text{findr } e) < i \wedge i < e\text{-length } e \longrightarrow e\text{-nth } e \ i = 0$
 using * *Greatest-le-nat*[of ?*P* - *e-length* *e*] **by** *fastforce*
qed

definition *r-findr* \equiv

let *g* =
 Cn 3 *r-ifz*
 [*Cn* 3 *r-nth* [*Id* 3 2, *Id* 3 0],
 Cn 3 *r-ifeq* [*Id* 3 0, *Id* 3 1, *Cn* 3 *S* [*Id* 3 0], *Id* 3 1],
 Id 3 0]
 in *Cn* 1 (*Pr* 1 *Z* *g*) [*Cn* 1 *r-length* [*Id* 1 0], *Id* 1 0]

lemma *r-findr-prim* [*simp*]: *prim-recfn* 1 *r-findr*
unfolding *r-findr-def* **by** *simp*

lemma *r-findr* [*simp*]: *eval* *r-findr* [*e*] = *findr* *e*

proof –

define *g* **where** *g* =
 Cn 3 *r-ifz*
 [*Cn* 3 *r-nth* [*Id* 3 2, *Id* 3 0],
 Cn 3 *r-ifeq* [*Id* 3 0, *Id* 3 1, *Cn* 3 *S* [*Id* 3 0], *Id* 3 1],
 Id 3 0]
then have *recfn* 3 *g*
 by *simp*
with *g-def* **have** *g*: *eval* *g* [*j*, *r*, *e*] $\downarrow =$
 (*if* *e-nth* *e* *j* $\neq 0$ **then** *j* **else** *if* *j* = *r* **then** *Suc* *j* **else** *r*) **for** *j* *r* *e*
 by *simp*
let ?*h* = *Pr* 1 *Z* *g*
have *recfn* 2 ?*h*
 by (*simp* *add*: $\langle \text{recfn } 3 \ g \rangle$)
let ?*P* = $\lambda e \ j \ i. \ i < j \wedge e\text{-nth } e \ i \neq 0$
let ?*G* = $\lambda e \ j. \ \text{Greatest } (?P \ e \ j)$
have *h*: *eval* ?*h* [*j*, *e*] =
 (*if* $\forall i < j. \ e\text{-nth } e \ i = 0$ **then** *Some* *j* **else** *Some* (?*G* *e* *j*)) **for** *j* *e*
proof (*induction* *j*)
 case 0
 then show ?*case* **using** $\langle \text{recfn } 2 \ ?h \rangle$ **by** *auto*
next
 case (*Suc* *j*)
 then have *eval* ?*h* [*Suc* *j*, *e*] = *eval* *g* [*j*, *the* (*eval* ?*h* [*j*, *e*]), *e*]
 using $\langle \text{recfn } 2 \ ?h \rangle$ **by** *auto*
 then have *eval* ?*h* [*Suc* *j*, *e*] =
 eval *g* [*j*, *if* $\forall i < j. \ e\text{-nth } e \ i = 0$ **then** *j* **else** ?*G* *e* *j*, *e*]
 using *Suc* **by** *auto*
 then have *: *eval* ?*h* [*Suc* *j*, *e*] $\downarrow =$
 (*if* *e-nth* *e* *j* $\neq 0$ **then** *j*
 else *if* *j* = (*if* $\forall i < j. \ e\text{-nth } e \ i = 0$ **then** *j* **else** ?*G* *e* *j*)
 then *Suc* *j*
 else (*if* $\forall i < j. \ e\text{-nth } e \ i = 0$ **then** *j* **else** ?*G* *e* *j*))
 using *g* **by** *simp*
 show ?*case*
 proof (*cases* $\forall i < \text{Suc } j. \ e\text{-nth } e \ i = 0$)
 case *True*
 then show ?*thesis* **using** * **by** *simp*
 next

```

case False
then have  $ex: \exists i < \text{Suc } j. e\text{-nth } e \ i \neq 0$ 
  by auto
show ?thesis
proof (cases  $e\text{-nth } e \ j = 0$ )
  case True
    then have  $ex': \exists i < j. e\text{-nth } e \ i \neq 0$ 
      using  $ex$  less-Suc-eq by fastforce
    then have ( $\text{if } \forall i < j. e\text{-nth } e \ i = 0 \text{ then } j \text{ else } ?G \ e \ j$ ) =  $?G \ e \ j$ 
      by metis
    moreover have  $?G \ e \ j < j$ 
      using  $ex'$  GreatestI-nat[of  $?P \ e \ j$ ] less-imp-le-nat by blast
    ultimately have  $\text{eval } ?h \ [\text{Suc } j, e] \downarrow = ?G \ e \ j$ 
      using  $*$  True by simp
    moreover have  $?G \ e \ j = ?G \ e \ (\text{Suc } j)$ 
      using True by (metis less-SucI less-Suc-eq)
    ultimately show ?thesis using  $ex$  by metis
  next
    case False
    then have  $\text{eval } ?h \ [\text{Suc } j, e] \downarrow = j$ 
      using  $*$  by simp
    moreover have  $?G \ e \ (\text{Suc } j) = j$ 
      using  $ex$  False Greatest-equality[of  $?P \ e \ (\text{Suc } j)$ ] by simp
    ultimately show ?thesis using  $ex$  by simp
  qed
qed
qed
let  $?hh = Cn \ 1 \ ?h \ [Cn \ 1 \ r\text{-length} \ [\text{Id } 1 \ 0], \ \text{Id } 1 \ 0]$ 
have  $\text{recfn } 1 \ ?hh$ 
  using  $\langle \text{recfn } 2 \ ?h \rangle$  by simp
with  $h$  have  $hh: \text{eval } ?hh \ [e] \downarrow =$ 
  ( $\text{if } \forall i < e\text{-length } e. e\text{-nth } e \ i = 0 \text{ then } e\text{-length } e \text{ else } ?G \ e \ (e\text{-length } e)$ ) for  $e$ 
  by auto
then have  $\text{eval } ?hh \ [e] = \text{findr } e \ \text{for } e$ 
  unfolding findr-def by auto
moreover have total  $?hh$ 
  using  $hh$  totalI1  $\langle \text{recfn } 1 \ ?hh \rangle$  by simp
ultimately show ?thesis
  using  $\langle \text{recfn } 1 \ ?hh \rangle$  g-def r-findr-def findr-def by metis
qed

lemma U0-in-CP:  $U_0 \in CP$ 
proof –
  define  $s$  where
     $s \equiv \lambda x. \text{if } \text{findr } x \downarrow = e\text{-length } x \text{ then } \text{Some } 0 \text{ else } \text{Some } (e\text{-take } (\text{Suc } (\text{the } (\text{findr } x)))) \ x$ 
  have  $s \in \mathcal{P}$ 
  proof –
    define  $r$  where
       $r \equiv Cn \ 1 \ r\text{-ifeq} \ [r\text{-findr}, \ r\text{-length}, \ Z, \ Cn \ 1 \ r\text{-take} \ [Cn \ 1 \ S \ [r\text{-findr}], \ \text{Id } 1 \ 0]]$ 
    then have  $\bigwedge x. \text{eval } r \ [x] = s \ x$ 
      using s-def findr-total by fastforce
    moreover have  $\text{recfn } 1 \ r$ 
      using r-def by simp
    ultimately show ?thesis by auto
  qed
moreover have learn-cp prenum  $U_0 \ s$ 

```

proof (rule learn-cpI)
show environment prenum U_0 s
using $\langle s \in \mathcal{P} \rangle$ s-def prenum-in-R2 U0-in-NUM **by** auto
show $\exists i. \text{prenum } i = f \wedge (\forall^\infty n. s (f \triangleright n) \downarrow = i)$ **if** $f \in U_0$ **for** f
proof (cases $f = (\lambda-. \text{Some } 0)$)
case True
then have $s (f \triangleright n) \downarrow = 0$ **for** n
using findr-def s-def **by** simp
then have $\forall n \geq 0. s (f \triangleright n) \downarrow = 0$ **by** simp
moreover have prenum $0 = f$
using True **by** auto
ultimately show ?thesis **by** auto
next
case False
then obtain ws **where** $ws: \text{length } ws > 0$ last $ws \neq 0$ $f = ws \odot 0^\infty$
using U0-def $\langle f \in U_0 \rangle$ almost0-canonical **by** blast
let $?m = \text{length } ws - 1$
let $?i = \text{list-encode } ws$
have prenum $?i = f$
using ws **by** auto
moreover have $s (f \triangleright n) \downarrow = ?i$ **if** $n \geq ?m$ **for** n
proof –
have $e\text{-nth } (f \triangleright n) ?m \neq 0$
using ws that **by** (simp add: last-conv-nth)
then have $\exists k < \text{Suc } n. e\text{-nth } (f \triangleright n) k \neq 0$
using le-imp-less-Suc that **by** blast
moreover have
 $(\text{GREATEST } k. k < e\text{-length } (f \triangleright n) \wedge e\text{-nth } (f \triangleright n) k \neq 0) = ?m$
proof (rule Greatest-equality)
show $?m < e\text{-length } (f \triangleright n) \wedge e\text{-nth } (f \triangleright n) ?m \neq 0$
using $\langle e\text{-nth } (f \triangleright n) ?m \neq 0 \rangle$ that **by** auto
show $\bigwedge y. y < e\text{-length } (f \triangleright n) \wedge e\text{-nth } (f \triangleright n) y \neq 0 \implies y \leq ?m$
using ws less-Suc-eq-le **by** fastforce
qed
ultimately have findr $(f \triangleright n) \downarrow = ?m$
using that findr-def **by** simp
moreover have $?m < e\text{-length } (f \triangleright n)$
using that **by** simp
ultimately have $s (f \triangleright n) \downarrow = e\text{-take } (\text{Suc } ?m) (f \triangleright n)$
using s-def **by** simp
moreover have $e\text{-take } (\text{Suc } ?m) (f \triangleright n) = \text{list-encode } ws$
proof –
have $\text{take } (\text{Suc } ?m) (\text{prefix } f n) = \text{prefix } f ?m$
using take-prefix[of $f ?m n$] ws that **by** (simp add: almost0-in-R1)
then have $\text{take } (\text{Suc } ?m) (\text{prefix } f n) = ws$
using ws prefixI **by** auto
then show ?thesis **by** simp
qed
ultimately show ?thesis **by** simp
qed
ultimately show ?thesis **by** auto
qed
show $\bigwedge f n. f \in U_0 \implies \text{prenum } (\text{the } (s (f \triangleright n))) \in U_0$
using U0-def **by** fastforce
qed
ultimately show ?thesis **using** CP-def **by** blast

qed

As a bit of an interlude, we can now show that CP is not closed under the subset relation. This works by removing functions from U_0 in a “noncomputable” way such that a strategy cannot ensure that every intermediate hypothesis is in that new class.

lemma *CP-not-closed-subseteq*: $\exists V U. V \subseteq U \wedge U \in CP \wedge V \notin CP$

proof –

– The numbering $g \in \mathcal{R}^2$ enumerates all functions $i0^\infty \in U_0$.

define g **where** $g \equiv \lambda i. [i] \odot 0^\infty$

have g -*inj*: $i = j$ **if** $g\ i = g\ j$ **for** $i\ j$

proof –

have $g\ i\ 0 \downarrow = i$ **and** $g\ j\ 0 \downarrow = j$

by (*simp-all add: g-def*)

with *that show* $i = j$

by (*metis option.inject*)

qed

– Define a class V . If the strategy φ_i learns g_i , it outputs a hypothesis for g_i on some shortest prefix g_i^m . Then the function $g_i^m 10^\infty$ is included in the class V ; otherwise g_i is included.

define V **where** $V \equiv$

{if learn-lim $\varphi\ \{g\ i\}$ ($\varphi\ i$)

then (*prefix* ($g\ i$) (*LEAST* $n. \varphi\ (\text{the } (\varphi\ i\ ((g\ i) \triangleright n))) = g\ i$)) @ [1] $\odot 0^\infty$

else $g\ i$ |

i. i $\in UNIV$ }

have $V \notin CP$ -*wrt* φ

proof

– Assuming $V \in CP_\varphi$, there is a CP strategy φ_i for V .

assume $V \in CP$ -*wrt* φ

then obtain s **where** $s: s \in \mathcal{P}$ *learn-cp* $\varphi\ V\ s$

using *CP-wrt-def learn-cpE(1)* **by** *auto*

then obtain i **where** $i: \varphi\ i = s$

using *phi-universal* **by** *auto*

show *False*

proof (*cases learn-lim* $\varphi\ \{g\ i\}$ ($\varphi\ i$))

case *learn: True*

– If φ_i learns g_i , it hypothesizes g_i on some shortest prefix g_i^m . Thus it hypothesizes g_i on some prefix of $g_i^m 10^\infty \in V$, too. But g_i is not a class-preserving hypothesis because $g_i \notin V$.

let $?P = \lambda n. \varphi\ (\text{the } (\varphi\ i\ ((g\ i) \triangleright n))) = g\ i$

let $?m = \text{Least } ?P$

have $\exists n. ?P\ n$

using $i\ s$ **by** (*meson learn infinite-hyp-wrong-not-Lim insertI1 lessI*)

then have $?P\ ?m$

using *LeastI-ex[of ?P]* **by** *simp*

define h **where** $h = (\text{prefix } (g\ i)\ ?m) @ [1] \odot 0^\infty$

then have $h \in V$

using *V-def learn* **by** *auto*

have $(g\ i) \triangleright ?m = h \triangleright ?m$

proof –

have *prefix* ($g\ i$) $?m = \text{prefix } h\ ?m$

unfolding h -*def* **by** (*simp add: prefix-prepend-less*)

then show $?thesis$ **by** *auto*

qed

then have $\varphi\ (\text{the } (\varphi\ i\ (h \triangleright ?m))) = g\ i$

using $\langle ?P\ ?m \rangle$ **by** *simp*

moreover have $g\ i \notin V$

proof
assume $g i \in V$
then obtain j **where** $j: g i =$
 (if learn-lim $\varphi \{g j\} (\varphi j)$
 then (prefix $(g j)$ (LEAST $n. \varphi$ (the $(\varphi j ((g j) \triangleright n))) = g j$)) @ $[1] \odot 0^\infty$
 else $g j$)
using V -def **by** *auto*
show *False*
proof (*cases learn-lim $\varphi \{g j\} (\varphi j)$*)
 case *True*
 then have $g i =$
 (prefix $(g j)$ (LEAST $n. \varphi$ (the $(\varphi j ((g j) \triangleright n))) = g j$)) @ $[1] \odot 0^\infty$
 (is $g i = ?vs$ @ $[1] \odot 0^\infty$)
 using j **by** *simp*
 moreover have $len: length ?vs > 0$ **by** *simp*
 ultimately have $g i (length ?vs) \downarrow = 1$
 by (*simp add: prepend-associative*)
 moreover have $g i (length ?vs) \downarrow = 0$
 using g -def len **by** *simp*
 ultimately show $?thesis$ **by** *simp*
 next
 case *False*
 then show $?thesis$
 using j g -inj learn **by** *auto*
 qed
qed
ultimately have φ (the $(\varphi i (h \triangleright ?m))) \notin V$ **by** *simp*
then have $\neg learn\text{-}cp \varphi V (\varphi i)$
 using $\langle h \in V \rangle learn\text{-}cpE(3)$ **by** *auto*
then show $?thesis$ **by** (*simp add: i s(2)*)
next
 — If φ_i does not learn g_i , then $g_i \in V$. Hence φ_i does not learn V .
 case *False*
 then have $g i \in V$
 using V -def **by** *auto*
 with *False* **have** $\neg learn\text{-}lim \varphi V (\varphi i)$
 using *learn-lim-closed-subseteq* **by** *auto*
 then show $?thesis$
 using $s(2)$ i **by** (*simp add: learn-cp-def*)
 qed
qed
then have $V \notin CP$
 using CP -wrt-phi **by** *simp*
moreover have $V \subseteq U_0$
 using V -def g -def U_0 -def **by** *auto*
ultimately show $?thesis$ **using** U_0 -in-CP **by** *auto*
qed

Continuing with the main result of this section, we show that U_0 cannot be learned finitely. Any FIN strategy would have to output a hypothesis for the constant zero function on some prefix. But U_0 contains infinitely many other functions starting with the same prefix, which the strategy then would not learn finitely.

lemma U_0 -not-in-FIN: $U_0 \notin FIN$

proof
assume $U_0 \in FIN$

then obtain ψ s **where** $\text{learn-fin } \psi \ U_0 \ s$
using FIN-def **by** blast
with learn-finE **have** $cp: \bigwedge f. f \in U_0 \implies$
 $\exists i \ n_0. \psi \ i = f \wedge (\forall n < n_0. s \ (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. s \ (f \triangleright n) \downarrow = \text{Suc } i)$
by simp-all

define z **where** $z = [] \odot 0^\infty$
then have $z \in U_0$
using U0-def **by** auto
with cp **obtain** $i \ n_0$ **where** $i: \psi \ i = z$ **and** $n0: \forall n \geq n_0. s \ (z \triangleright n) \downarrow = \text{Suc } i$
by blast

define w **where** $w = \text{replicate } (\text{Suc } n_0) \ 0 \ @ \ [1] \odot 0^\infty$
then have $\text{prefix } w \ n_0 = \text{replicate } (\text{Suc } n_0) \ 0$
by $(\text{simp add: prefix-prepend-less})$
moreover have $\text{prefix } z \ n_0 = \text{replicate } (\text{Suc } n_0) \ 0$
using $\text{prefixI[of replicate } (\text{Suc } n_0) \ 0 \ z] \ \text{less-Suc-eq-0-disj}$ **unfolding** $z\text{-def}$
by fastforce
ultimately have $z \triangleright \ n_0 = w \triangleright \ n_0$
by $(\text{simp add: init-prefixE})$
with $n0$ **have** $*$: $s \ (w \triangleright \ n_0) \downarrow = \text{Suc } i$ **by** auto

have $w \in U_0$ **using** $w\text{-def}$ U0-def **by** auto
with cp **obtain** $i' \ n_0'$ **where** $i': \psi \ i' = w$
and $n0': \forall n < n_0'. s \ (w \triangleright \ n) \downarrow = 0 \ \forall n \geq n_0'. s \ (w \triangleright \ n) \downarrow = \text{Suc } i'$
by blast

have $i \neq i'$
proof
assume $i = i'$
then have $w = z$
using $i \ i'$ **by** simp
have $w \ (\text{Suc } n_0) \downarrow = 1$
using $w\text{-def}$ $\text{prepend[of replicate } (\text{Suc } n_0) \ 0 \ @ \ [1] \ 0^\infty \ \text{Suc } n_0]$
by $(\text{metis length-append-singleton length-replicate lessI nth-append-length})$
moreover have $z \ (\text{Suc } n_0) \downarrow = 0$
using $z\text{-def}$ **by** simp
ultimately show False
using $\langle w = z \rangle$ **by** simp
qed
then have $s \ (w \triangleright \ n_0) \downarrow \neq \text{Suc } i$
using $n0'$ **by** $(\text{cases } n_0 < n_0') \ \text{simp-all}$
with $*$ **show** False **by** simp
qed

theorem $\text{FIN-subset-CP: } \text{FIN} \subset \text{CP}$
using U0-in-CP U0-not-in-FIN FIN-subseteq-CP **by** auto

2.4 NUM and FIN are incomparable

The class V_0 of all total recursive functions f where $f(0)$ is a Gödel number of f can be learned finitely by always hypothesizing $f(0)$. The class is not in NUM and therefore serves to separate NUM and FIN.

definition $V0 :: \text{partial1 set } (\langle V_0 \rangle)$ **where**

$V_0 = \{f. f \in \mathcal{R} \wedge \varphi (the (f 0)) = f\}$

lemma *V0-altdef*: $V_0 = \{[i] \odot f \mid i f. f \in \mathcal{R} \wedge \varphi i = [i] \odot f\}$
(is $V_0 = ?W$)

proof

show $V_0 \subseteq ?W$

proof

fix f

assume $f \in V_0$

then have $f \in \mathcal{R}$

unfolding *V0-def* **by** *simp*

then obtain i **where** $i: f 0 \Downarrow i$ **by** *fastforce*

define g **where** $g = (\lambda x. f (x + 1))$

then have $g \in \mathcal{R}$

using *skip-R1* [*OF* $\langle f \in \mathcal{R} \rangle$] **by** *blast*

moreover have $[i] \odot g = f$

using *g-def* i **by** *auto*

moreover have $\varphi i = f$

using $\langle f \in V_0 \rangle$ *V0-def* i **by** *force*

ultimately show $f \in ?W$ **by** *auto*

qed

show $?W \subseteq V_0$

proof

fix g

assume $g \in ?W$

then have $\varphi (the (g 0)) = g$ **by** *auto*

moreover have $g \in \mathcal{R}$

using *prepend-in-R1* $\langle g \in ?W \rangle$ **by** *auto*

ultimately show $g \in V_0$

by (*simp add: V0-def*)

qed

qed

lemma *V0-in-FIN*: $V_0 \in FIN$

proof –

define s **where** $s = (\lambda x. Some (Suc (e-hd x)))$

have $s \in \mathcal{P}$

proof –

define r **where** $r = Cn 1 S [r-hd]$

then have *recfn* 1 r **by** *simp*

moreover have *eval* $r [x] \Downarrow Suc (e-hd x)$ **for** x

unfolding *r-def* **by** *simp*

ultimately show *?thesis*

using *s-def* **by** *blast*

qed

have $s: s (f \triangleright n) \Downarrow Suc (the (f 0))$ **for** $f n$

unfolding *s-def* **by** *simp*

have *learn-fin* $\varphi V_0 s$

proof (*rule learn-finI*)

show *environment* $\varphi V_0 s$

using *s-def* $\langle s \in \mathcal{P} \rangle$ *phi-in-P2* *V0-def* **by** *auto*

show $\exists i n_0. \varphi i = f \wedge (\forall n < n_0. s (f \triangleright n) \Downarrow 0) \wedge (\forall n \geq n_0. s (f \triangleright n) \Downarrow Suc i)$

if $f \in V_0$ **for** f

using *that V0-def* s **by** *auto*

qed

then show *?thesis* **using** *FIN-def* **by** *auto*

qed

To every $f \in \mathcal{R}$ a number can be prepended that is a Gödel number of the resulting function. Such a function is then in V_0 .

If V_0 was in NUM, it would be embedded in a total numbering. Shifting this numbering to the left, essentially discarding the values at point 0, would yield a total numbering for \mathcal{R} , which contradicts *R1-not-in-NUM*. This proves $V_0 \notin \text{NUM}$.

lemma *prepend-goedel*:

assumes $f \in \mathcal{R}$

shows $\exists i. \varphi i = [i] \odot f$

proof –

obtain r where $r: \text{recfn } 1 \ r \ \text{total } r \ \wedge x. \text{eval } r \ [x] = f \ x$

using *assms* by *auto*

define $r\text{-psi}$ where $r\text{-psi} = \text{Cn } 2 \ r\text{-ifz } [\text{Id } 2 \ 1, \ \text{Id } 2 \ 0, \ \text{Cn } 2 \ r \ [\text{Cn } 2 \ r\text{-dec } [\text{Id } 2 \ 1]]]$

then have $\text{recfn } 2 \ r\text{-psi}$

using $r(1)$ by *simp*

have $\text{eval } r\text{-psi } [i, x] = (\text{if } x = 0 \ \text{then } \text{Some } i \ \text{else } f \ (x - 1))$ for $i \ x$

proof –

have $\text{eval } (\text{Cn } 2 \ r \ [\text{Cn } 2 \ r\text{-dec } [\text{Id } 2 \ 1]]) [i, x] = f \ (x - 1)$

using r by *simp*

then have $\text{eval } r\text{-psi } [i, x] = \text{eval } r\text{-ifz } [x, i, \text{the } (f \ (x - 1))]$

unfolding $r\text{-psi-def}$ using $\langle \text{recfn } 2 \ r\text{-psi} \rangle \ r \ R1\text{-imp-total1}[\text{OF } \text{assms}]$ by *auto*

then show *?thesis*

using *assms* by *simp*

qed

with $\langle \text{recfn } 2 \ r\text{-psi} \rangle$ have $(\lambda i \ x. \text{if } x = 0 \ \text{then } \text{Some } i \ \text{else } f \ (x - 1)) \in \mathcal{P}^2$

by *auto*

with *kleene-fixed-point* obtain i where

$\varphi i = (\lambda x. \text{if } x = 0 \ \text{then } \text{Some } i \ \text{else } f \ (x - 1))$

by *blast*

then have $\varphi i = [i] \odot f$ by *auto*

then show *?thesis* by *auto*

qed

lemma *V0-in-FIN-minus-NUM*: $V_0 \in \text{FIN} - \text{NUM}$

proof –

have $V_0 \notin \text{NUM}$

proof

assume $V_0 \in \text{NUM}$

then obtain ψ where $\psi: \psi \in \mathcal{R}^2 \ \wedge f. f \in V_0 \implies \exists i. \psi i = f$

by *auto*

define ψ' where $\psi' i \ x = \psi i \ (\text{Suc } x)$ for $i \ x$

have $\psi' \in \mathcal{R}^2$

proof

from $\psi(1)$ obtain $r\text{-psi}$ where

$r\text{-psi}: \text{recfn } 2 \ r\text{-psi} \ \text{total } r\text{-psi} \ \wedge i \ x. \text{eval } r\text{-psi } [i, x] = \psi i \ x$

by *blast*

define $r\text{-psi}'$ where $r\text{-psi}' = \text{Cn } 2 \ r\text{-psi} \ [\text{Id } 2 \ 0, \ \text{Cn } 2 \ S \ [\text{Id } 2 \ 1]]$

then have $\text{recfn } 2 \ r\text{-psi}'$ and $\wedge i \ x. \text{eval } r\text{-psi}' [i, x] = \psi' i \ x$

unfolding $r\text{-psi}'\text{-def}$ $\psi'\text{-def}$ using $r\text{-psi}$ by *simp-all*

then show $\psi' \in \mathcal{P}^2$ by *blast*

show *total2* ψ'

using $\psi'\text{-def}$ $\psi(1)$ by (*simp add: total2I*)

qed

have $\exists i. \psi' i = f$ if $f \in \mathcal{R}$ for f

proof –
from *that* **obtain** j **where** $j: \varphi j = [j] \odot f$
using *prepend-goedel* **by** *auto*
then **have** $\varphi j \in V_0$
using *that V0-altdef* **by** *auto*
with ψ **obtain** i **where** $\psi i = \varphi j$ **by** *auto*
then **have** $\psi' i = f$
using ψ' -*def j* **by** (*auto simp add: prepend-at-ge*)
then **show** *?thesis* **by** *auto*
qed
with $\langle \psi' \in \mathcal{R}^2 \rangle$ **have** $\mathcal{R} \in \text{NUM}$ **by** *auto*
with *R1-not-in-NUM* **show** *False* **by** *simp*
qed
then **show** *?thesis*
using *V0-in-FIN* **by** *auto*
qed

corollary *FIN-not-subseteq-NUM*: $\neg \text{FIN} \subseteq \text{NUM}$
using *V0-in-FIN-minus-NUM* **by** *auto*

2.5 NUM and CP are incomparable

There are FIN classes outside of NUM, and CP encompasses FIN. Hence there are CP classes outside of NUM, too.

theorem *CP-not-subseteq-NUM*: $\neg \text{CP} \subseteq \text{NUM}$
using *FIN-subseteq-CP FIN-not-subseteq-NUM* **by** *blast*

Conversely there is a subclass of U_0 that is in NUM but cannot be learned in a class-preserving way. The following proof is due to Jantke and Beick [10]. The idea is to diagonalize against all strategies, that is, all partial recursive functions.

theorem *NUM-not-subseteq-CP*: $\neg \text{NUM} \subseteq \text{CP}$

proof –

– Define a family of functions f_k .
define f **where** $f \equiv \lambda k. [k] \odot 0^\infty$
then **have** $f k \in \mathcal{R}$ **for** k
using *almost0-in-R1* **by** *auto*

– If the strategy φ_k learns f_k it hypothesizes f_k for some shortest prefix $f_k^{a_k}$. Define functions $f'_k = k0^{a_k}10^\infty$.

define a **where**
 $a \equiv \lambda k. \text{LEAST } x. (\varphi (\text{the } ((\varphi k) ((f k) \triangleright x)))) = f k$
define f' **where** $f' \equiv \lambda k. (k \# (\text{replicate } (a k) 0) @ [1]) \odot 0^\infty$
then **have** $f' k \in \mathcal{R}$ **for** k
using *almost0-in-R1* **by** *auto*

– Although f_k and f'_k differ, they share the prefix of length $a_k + 1$.

have *init-eq*: $(f' k) \triangleright (a k) = (f k) \triangleright (a k)$ **for** k

proof (*rule init-eqI*)

fix x **assume** $x \leq a k$

then **show** $f' k x = f k x$

by (*cases x = 0*) (*simp-all add: nth-append f'-def f-def*)

qed

have $f k \neq f' k$ **for** k

proof –

have $f k (Suc (a k)) \downarrow = 0$ **using** $f\text{-def}$ **by** $auto$
moreover have $f' k (Suc (a k)) \downarrow = 1$
using $f'\text{-def}$ $prepend[of (k \# (replicate (a k) 0) @ [1]) 0^\infty Suc (a k)]$
by ($metis\ length\ Cons\ length\ append\ singleton\ length\ replicate\ lessI\ nth\ Cons\ Suc\ nth\ append\ length$)
ultimately show $?thesis$ **by** $auto$
qed

— The separating class U contains f'_k if φ_k learns f_k ; otherwise it contains f_k .

define U **where**

$U \equiv \{if\ learn\text{-}lim\ \varphi\ \{f\ k\}\ (\varphi\ k)\ then\ f'\ k\ else\ f\ k\ |k.\ k \in UNIV\}$

have $U \notin CP$

proof

assume $U \in CP$

have $\exists k.\ learn\text{-}cp\ \varphi\ U\ (\varphi\ k)$

proof —

have $\exists \psi\ s.\ learn\text{-}cp\ \psi\ U\ s$

using $CP\text{-}def\ \langle U \in CP \rangle$ **by** $auto$

then obtain s **where** $s:\ learn\text{-}cp\ \varphi\ U\ s$

using $learn\text{-}cp\text{-}wrt\text{-}goedel[OF\ goedel\text{-}numbering\text{-}phi]$ **by** $blast$

then obtain k **where** $\varphi\ k = s$

using $phi\text{-}universal\ learn\text{-}cp\text{-}def\ learn\text{-}lim\text{-}def$ **by** $auto$

then show $?thesis$ **using** s **by** $auto$

qed

then obtain k **where** $k:\ learn\text{-}cp\ \varphi\ U\ (\varphi\ k)$ **by** $auto$

then have $learn:\ learn\text{-}lim\ \varphi\ U\ (\varphi\ k)$

using $learn\text{-}cp\text{-}def$ **by** $simp$

— If f_k was in U , φ_k would learn it. But then, by definition of U , f_k would not be in U .

Hence $f_k \notin U$.

have $f k \notin U$

proof

assume $f k \in U$

then obtain m **where** $m:\ f k = (if\ learn\text{-}lim\ \varphi\ \{f\ m\}\ (\varphi\ m)\ then\ f'\ m\ else\ f\ m)$

using $U\text{-}def$ **by** $auto$

have $f k 0 \downarrow = m$

using $f\text{-}def\ f'\text{-}def\ m$ **by** $simp$

moreover have $f k 0 \downarrow = k$ **by** ($simp\ add:\ f\text{-}def$)

ultimately have $m = k$ **by** $simp$

with m **have** $f k = (if\ learn\text{-}lim\ \varphi\ \{f\ k\}\ (\varphi\ k)\ then\ f'\ k\ else\ f\ k)$

by $auto$

moreover have $learn\text{-}lim\ \varphi\ \{f\ k\}\ (\varphi\ k)$

using $\langle f k \in U \rangle\ learn\text{-}lim\text{-}closed\text{-}subsetq[OF\ learn]$ **by** $simp$

ultimately have $f k = f' k$

by $simp$

then show $False$

using $\langle f k \neq f' k \rangle$ **by** $simp$

qed

then have $f' k \in U$ **using** $U\text{-}def$ **by** $fastforce$

then have $in\text{-}U:\ \forall n.\ \varphi\ (the\ ((\varphi\ k)\ ((f' k) \triangleright n))) \in U$

using $learn\text{-}cpE(3)[OF\ k]$ **by** $simp$

— Since $f'_k \in U$, the strategy φ_k learns f_k . Then a_k is well-defined, $f'^{a_k} = f^{a_k}$, and φ_k hypothesizes f_k on f'^{a_k} , which is not a class-preserving hypothesis.

have $learn\text{-}lim\ \varphi\ \{f\ k\}\ (\varphi\ k)$ **using** $U\text{-}def\ \langle f k \notin U \rangle$ **by** $fastforce$

then have $\exists i\ n_0.\ \varphi\ i = f k \wedge (\forall n \geq n_0.\ \varphi\ k\ ((f k) \triangleright n) \downarrow = i)$

using $learn\text{-}limE(2)$ **by** $simp$

then obtain $i\ n_0$ **where** $\varphi\ i = f\ k \wedge (\forall n \geq n_0. \varphi\ k\ ((f\ k) \triangleright n) \downarrow = i)$
by *auto*
then have $\varphi\ (\text{the } (\varphi\ k\ ((f\ k) \triangleright (a\ k)))) = f\ k$
using *a-def LeastI*[of $\lambda x. (\varphi\ (\text{the } ((\varphi\ k)\ ((f\ k) \triangleright x)))) = f\ k\ n_0]$
by *simp*
then have $\varphi\ (\text{the } ((\varphi\ k)\ ((f'\ k) \triangleright (a\ k)))) = f\ k$
using *init-eq* **by** *simp*
then show *False*
using $\langle f\ k \notin U \rangle$ *in-U* **by** *metis*
qed
moreover have $U \in \text{NUM}$
using *NUM-closed-subseteq*[*OF U0-in-NUM*, of U] *f-def f'-def U0-def U-def*
by *fastforce*
ultimately show *?thesis* **by** *auto*
qed

2.6 NUM is a proper subset of TOTAL

A NUM class U is embedded in a total numbering ψ . The strategy S with $S(f^n) = \min\{i \mid \forall k \leq n : \psi_i(k) = f(k)\}$ for $f \in U$ converges to the least index of f in ψ , and thus learns f in the limit. Moreover it will be a TOTAL strategy because ψ contains only total functions. This shows $\text{NUM} \subseteq \text{TOTAL}$.

First we define, for every hypothesis space ψ , a function that tries to determine for a given list e and index i whether e is a prefix of ψ_i . In other words it tries to decide whether i is a consistent hypothesis for e . “Tries” refers to the fact that the function will diverge if $\psi_i(x) \uparrow$ for any $x \leq |e|$. We start with a version that checks the list only up to a given length.

definition *r-consist-upto* :: *recf* \Rightarrow *recf* **where**

r-consist-upto r-psi \equiv
let $g = \text{Cn } 4\ r\text{-ifeq}$
 $[\text{Cn } 4\ r\text{-psi } [\text{Id } 4\ 2, \text{Id } 4\ 0], \text{Cn } 4\ r\text{-nth } [\text{Id } 4\ 3, \text{Id } 4\ 0], \text{Id } 4\ 1, r\text{-constn } 3\ 1]$
in *Pr* 2 (*r-constn* 1 0) g

lemma *r-consist-upto-recfn*: *recfn* 2 *r-psi* \Longrightarrow *recfn* 3 (*r-consist-upto r-psi*)
using *r-consist-upto-def* **by** *simp*

lemma *r-consist-upto*:

assumes *recfn* 2 *r-psi*
shows $\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow \Longrightarrow$
 $\text{eval } (r\text{-consist-upto } r\text{-psi}) [j, i, e] =$
(if $\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow = e\text{-nth } e\ k$ *then* *Some* 0 *else* *Some* 1)
and $\neg (\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow) \Longrightarrow \text{eval } (r\text{-consist-upto } r\text{-psi}) [j, i, e] \uparrow$

proof –

define g **where** $g =$
 $\text{Cn } 4\ r\text{-ifeq}$
 $[\text{Cn } 4\ r\text{-psi } [\text{Id } 4\ 2, \text{Id } 4\ 0], \text{Cn } 4\ r\text{-nth } [\text{Id } 4\ 3, \text{Id } 4\ 0], \text{Id } 4\ 1, r\text{-constn } 3\ 1]$
then have *recfn* 4 g
using *assms* **by** *simp*
moreover have $\text{eval } (\text{Cn } 4\ r\text{-nth } [\text{Id } 4\ 3, \text{Id } 4\ 0]) [j, r, i, e] \downarrow = e\text{-nth } e\ j$ **for** $j\ r\ i\ e$
by *simp*
moreover have $\text{eval } (r\text{-constn } 3\ 1) [j, r, i, e] \downarrow = 1$ **for** $j\ r\ i\ e$
by *simp*
moreover have $\text{eval } (\text{Cn } 4\ r\text{-psi } [\text{Id } 4\ 2, \text{Id } 4\ 0]) [j, r, i, e] = \text{eval } r\text{-psi } [i, j]$ **for** $j\ r\ i\ e$

using *assms(1)* **by** *simp*
ultimately have $g: \text{eval } g [j, r, i, e] =$
(if eval r-psi [i, j] \uparrow then None
else if eval r-psi [i, j] $\downarrow = e\text{-nth } e \text{ } j$ then Some r else Some 1)
for $j \ r \ i \ e$
using $\langle \text{recfn } 4 \ g \rangle$ *g-def assms* **by** *auto*
have goal1: $\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow \implies$
 $\text{eval } (r\text{-consist-upto } r\text{-psi}) [j, i, e] =$
(if $\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow = e\text{-nth } e \text{ } k$ then Some 0 else Some 1)
for $j \ i \ e$
proof (*induction j*)
case 0
then show *?case*
using *r-consist-upto-def r-consist-upto-recfn assms eval-Pr-0* **by** *simp*
next
case (*Suc j*)
then have $\text{eval } (r\text{-consist-upto } r\text{-psi}) [\text{Suc } j, i, e] =$
 $\text{eval } g [j, \text{the } (\text{eval } (r\text{-consist-upto } r\text{-psi}) [j, i, e]), i, e]$
using *assms eval-Pr-converg-Suc g-def r-consist-upto-def r-consist-upto-recfn*
by *simp*
also have $\dots = \text{eval } g [j, \text{if } \forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow = e\text{-nth } e \text{ } k \text{ then } 0 \text{ else } 1, i, e]$
using *Suc* **by** *auto*
also have $\dots \downarrow = (\text{if eval r-psi [i, j] } \downarrow = e\text{-nth } e \text{ } j$
 $\text{then if } \forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow = e\text{-nth } e \text{ } k \text{ then } 0 \text{ else } 1 \text{ else } 1)$
using *g* **by** (*simp add: Suc.prem*s)
also have $\dots \downarrow = (\text{if } \forall k < \text{Suc } j. \text{eval } r\text{-psi } [i, k] \downarrow = e\text{-nth } e \text{ } k \text{ then } 0 \text{ else } 1)$
by (*simp add: less-Suc-eq*)
finally show *?case* **by** *simp*
qed
then show $\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow \implies$
 $\text{eval } (r\text{-consist-upto } r\text{-psi}) [j, i, e] =$
(if $\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow = e\text{-nth } e \text{ } k$ then Some 0 else Some 1)
by *simp*
show $\neg (\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow) \implies \text{eval } (r\text{-consist-upto } r\text{-psi}) [j, i, e] \uparrow$
proof –
assume $\neg (\forall k < j. \text{eval } r\text{-psi } [i, k] \downarrow)$
then have $\exists k < j. \text{eval } r\text{-psi } [i, k] \uparrow$ **by** *simp*
let $?P = \lambda k. k < j \wedge \text{eval } r\text{-psi } [i, k] \uparrow$
define *kmin* **where** $kmin = \text{Least } ?P$
then have $?P \ kmin$
using *LeastI-ex[of ?P] $\langle \exists k < j. \text{eval } r\text{-psi } [i, k] \uparrow \rangle$* **by** *auto*
from *kmin-def* **have** $\bigwedge k. k < kmin \implies \neg ?P \ k$
using *kmin-def not-less-Least[of - ?P]* **by** *blast*
then have $\forall k < kmin. \text{eval } r\text{-psi } [i, k] \downarrow$
using $\langle ?P \ kmin \rangle$ **by** *simp*
then have $\text{eval } (r\text{-consist-upto } r\text{-psi}) [kmin, i, e] =$
(if $\forall k < kmin. \text{eval } r\text{-psi } [i, k] \downarrow = e\text{-nth } e \text{ } k$ then Some 0 else Some 1)
using *goal1* **by** *simp*
moreover have $\text{eval } r\text{-psi } [i, kmin] \uparrow$
using $\langle ?P \ kmin \rangle$ **by** *simp*
ultimately have $\text{eval } (r\text{-consist-upto } r\text{-psi}) [\text{Suc } kmin, i, e] \uparrow$
using *r-consist-upto-def g assms* **by** *simp*
moreover have $j \geq kmin$
using $\langle ?P \ kmin \rangle$ **by** *simp*
ultimately show $\text{eval } (r\text{-consist-upto } r\text{-psi}) [j, i, e] \uparrow$
using *r-consist-upto-def r-consist-upto-recfn $\langle ?P \ kmin \rangle$ eval-Pr-converg-le assms*

by (metis (full-types) Suc-leI length-Cons list.size(3) numeral-2-eq-2 numeral-3-eq-3)
qed
qed

The next function provides the consistency decision functions we need.

definition *consistent* :: *partial2* \Rightarrow *partial2* **where**

consistent ψ *i e* \equiv
if $\forall k < e\text{-length } e. \psi$ *i k* \downarrow
then if $\forall k < e\text{-length } e. \psi$ *i k* $\downarrow = e\text{-nth } e$ *k*
then *Some 0* else *Some 1*
else *None*

Given *i* and *e*, *consistent* ψ decides whether *e* is a prefix of ψ_i , provided ψ_i is defined for the length of *e*.

definition *r-consistent* :: *recf* \Rightarrow *recf* **where**

r-consistent *r-psi* \equiv
Cn 2 (*r-consist-upto* *r-psi*) [Cn 2 *r-length* [Id 2 1], Id 2 0, Id 2 1]

lemma *r-consistent-recfn* [*simp*]: *recfn* 2 *r-psi* \Longrightarrow *recfn* 2 (*r-consistent* *r-psi*)
using *r-consistent-def* *r-consist-upto-recfn* **by** *simp*

lemma *r-consistent-converg*:

assumes *recfn* 2 *r-psi* **and** $\forall k < e\text{-length } e. \text{eval } r\text{-psi } [i, k] \downarrow$
shows $\text{eval } (r\text{-consistent } r\text{-psi}) [i, e] \downarrow =$
($\text{if } \forall k < e\text{-length } e. \text{eval } r\text{-psi } [i, k] \downarrow = e\text{-nth } e$ *k* then 0 else 1)

proof –

have $\text{eval } (r\text{-consistent } r\text{-psi}) [i, e] = \text{eval } (r\text{-consist-upto } r\text{-psi}) [e\text{-length } e, i, e]$
using *r-consistent-def* *r-consist-upto-recfn* *assms*(1) **by** *simp*
then show *?thesis* **using** *assms* *r-consist-upto*(1) **by** *simp*

qed

lemma *r-consistent-diverg*:

assumes *recfn* 2 *r-psi* **and** $\exists k < e\text{-length } e. \text{eval } r\text{-psi } [i, k] \uparrow$
shows $\text{eval } (r\text{-consistent } r\text{-psi}) [i, e] \uparrow$
unfolding *r-consistent-def*
using *r-consist-upto-recfn*[*OF* *assms*(1)] *r-consist-upto*[*OF* *assms*(1)] *assms*(2)
by *simp*

lemma *r-consistent*:

assumes *recfn* 2 *r-psi* **and** $\forall x y. \text{eval } r\text{-psi } [x, y] = \psi$ *x y*
shows $\text{eval } (r\text{-consistent } r\text{-psi}) [i, e] = \text{consistent } \psi$ *i e*

proof (*cases* $\forall k < e\text{-length } e. \psi$ *i k* \downarrow)

case *True*

then have $\forall k < e\text{-length } e. \text{eval } r\text{-psi } [i, k] \downarrow$
using *assms* **by** *simp*

then show *?thesis*

unfolding *consistent-def* **using** *True* **by** (*simp* *add: assms* *r-consistent-converg*)

next

case *False*

then have $\text{consistent } \psi$ *i e* \uparrow

unfolding *consistent-def* **by** *auto*

moreover have $\text{eval } (r\text{-consistent } r\text{-psi}) [i, e] \uparrow$

using *r-consistent-diverg*[*OF* *assms*(1)] *assms* *False* **by** *simp*

ultimately show *?thesis* **by** *simp*

qed

lemma *consistent-in-P2*:
assumes $\psi \in \mathcal{P}^2$
shows *consistent* $\psi \in \mathcal{P}^2$
using *assms r-consistent P2E[OF assms(1)] P2I r-consistent-recfn* **by** *metis*

lemma *consistent-for-R2*:
assumes $\psi \in \mathcal{R}^2$
shows *consistent* ψ *i e =*
*(if $\forall j < e\text{-length } e. \psi$ $i j \downarrow = e\text{-nth } e j$ then *Some 0* else *Some 1*)*
using *assms* **by** (*simp add: consistent-def*)

lemma *consistent-init*:
assumes $\psi \in \mathcal{R}^2$ **and** $f \in \mathcal{R}$
shows *consistent* ψ *i (f \triangleright n) = (if ψ $i \triangleright n = f \triangleright n$ then *Some 0* else *Some 1*)*
using *consistent-def[of - - init f n] assms init-eq-iff-eq-upto* **by** *simp*

lemma *consistent-in-R2*:
assumes $\psi \in \mathcal{R}^2$
shows *consistent* $\psi \in \mathcal{R}^2$
using *total2I consistent-in-P2 consistent-for-R2[OF assms] P2-total-imp-R2 R2-imp-P2 assms*
by (*metis option.simps(3)*)

For total hypothesis spaces the next function computes the minimum hypothesis consistent with a given prefix. It diverges if no such hypothesis exists.

definition *min-cons-hyp* :: *partial2* \Rightarrow *partial1* **where**
min-cons-hyp ψ $e \equiv$
*if $\exists i. \text{consistent } \psi$ $i e \downarrow = 0$ then *Some (LEAST i. consistent ψ $i e \downarrow = 0$)* else *None**

lemma *min-cons-hyp-in-P1*:
assumes $\psi \in \mathcal{R}^2$
shows *min-cons-hyp* $\psi \in \mathcal{P}$
proof –
from *assms consistent-in-R2* **obtain** rc **where**
 $rc: \text{recfn } 2 \text{ } rc \text{ total } rc \wedge i e. \text{eval } rc [i, e] = \text{consistent } \psi$ $i e$
using *R2E[of consistent ψ]* **by** *metis*
define r **where** $r = Mn$ 1 rc
then have *recfn* 1 r
using *rc(1)* **by** *simp*
moreover from *this* **have** *eval* $r [e] = \text{min-cons-hyp } \psi$ e **for** e
using *r-def eval-Mn'[of 1 rc [e]] rc min-cons-hyp-def assms*
by (*auto simp add: consistent-in-R2*)
ultimately show *?thesis* **by** *auto*
qed

The function *min-cons-hyp* ψ is a strategy for learning all NUM classes embedded in ψ . It is an example of an “identification-by-enumeration” strategy.

lemma *NUM-imp-learn-total*:
assumes $\psi \in \mathcal{R}^2$ **and** $U \in \text{NUM-wrt } \psi$
shows *learn-total* ψ U (*min-cons-hyp* ψ)
proof (*rule learn-totalI*)
have *ex-psi-i-f: $\exists i. \psi$ $i = f$ if $f \in U$* **for** f
using *assms that NUM-wrt-def* **by** *simp*
moreover have *consistent-eq-0: consistent* ψ $i ((\psi$ $i) \triangleright n) \downarrow = 0$ **for** $i n$
using *assms* **by** (*simp add: consistent-init*)

ultimately have $\bigwedge f n. f \in U \implies \text{min-cons-hyp } \psi (f \triangleright n) \downarrow$
 using *min-cons-hyp-def* *assms(1)* by *fastforce*
 then show *env: environment* ψU (*min-cons-hyp* ψ)
 using *assms* *NUM-wrt-def* *min-cons-hyp-in-P1* *NUM-E(1)* *NUM-I* by *auto*

show $\bigwedge f n. f \in U \implies \psi$ (*the* (*min-cons-hyp* $\psi (f \triangleright n)$)) $\in \mathcal{R}$
 using *assms* by (*simp*)

show $\exists i. \psi i = f \wedge (\forall^\infty n. \text{min-cons-hyp } \psi (f \triangleright n) \downarrow = i)$ if $f \in U$ for f
 proof –

from *that env* have $f \in \mathcal{R}$ by *auto*

let $?P = \lambda i. \psi i = f$

define *imin* where *imin* $\equiv \text{Least } ?P$

with *ex-psi-i-f* that have *imin*: $?P \text{ } i \text{min} \bigwedge j. ?P j \implies j \geq i \text{min}$

using *LeastI-ex[of ?P]* *Least-le[of ?P]* by *simp-all*

then have *f-neq*: $\psi i \neq f$ if $i < i \text{min}$ for i

using *leD* that by *auto*

let $?Q = \lambda i n. \psi i \triangleright n \neq f \triangleright n$

define *nu* :: $\text{nat} \Rightarrow \text{nat}$ where *nu* = $(\lambda i. \text{SOME } n. ?Q i n)$

have *nu-neq*: $\psi i \triangleright (\text{nu } i) \neq f \triangleright (\text{nu } i)$ if $i < i \text{min}$ for i

proof –

from *assms* have $\psi i \in \mathcal{R}$ by *simp*

moreover from *assms* *imin(1)* have $f \in \mathcal{R}$ by *auto*

moreover have $f \neq \psi i$

using *that f-neq* by *auto*

ultimately have $\exists n. f \triangleright n \neq (\psi i) \triangleright n$

using *neq-fun-neq-init* by *simp*

then show $?Q i (\text{nu } i)$

unfolding *nu-def* using *someI-ex[of $\lambda n. ?Q i n$]* by *metis*

qed

have $\exists n_0. \forall n \geq n_0. \text{min-cons-hyp } \psi (f \triangleright n) \downarrow = i \text{min}$

proof (*cases* *imin = 0*)

case *True*

then have $\forall n. \text{min-cons-hyp } \psi (f \triangleright n) \downarrow = i \text{min}$

using *consistent-eq-0* *assms(1)* *imin(1)* *min-cons-hyp-def* by *auto*

then show *?thesis* by *simp*

next

case *False*

define n_0 where $n_0 = \text{Max} (\text{set} (\text{map } \text{nu } [0..<i \text{min}]))$ (*is* $- = \text{Max } ?N$)

have $\text{nu } i \leq n_0$ if $i < i \text{min}$ for i

proof –

have *finite* $?N$

using *n₀-def* by *simp*

moreover have $?N \neq \{\}$

using *False n₀-def* by *simp*

moreover have $\text{nu } i \in ?N$

using *that* by *simp*

ultimately show *?thesis*

using *that Max-ge n₀-def* by *blast*

qed

then have $\psi i \triangleright n_0 \neq f \triangleright n_0$ if $i < i \text{min}$ for i

using *nu-neq neq-init-forall-ge* that by *blast*

then have $*$: $\psi i \triangleright n \neq f \triangleright n$ if $i < i \text{min}$ and $n \geq n_0$ for $i n$

```

    using nu-neq neq-init-forall-ge that by blast

have  $\psi \text{ imin} \triangleright n = f \triangleright n$  for  $n$ 
  using imin(1) by simp
moreover have (consistent  $\psi \ i \ (f \triangleright n) \ \downarrow = 0) = (\psi \ i \triangleright n = f \triangleright n)$  for  $i \ n$ 
  by (simp add:  $\langle f \in \mathcal{R} \rangle \text{ assms}(1) \text{ consistent-init}$ )
ultimately have min-cons-hyp  $\psi \ (f \triangleright n) \ \downarrow = (\text{LEAST } i. \psi \ i \triangleright n = f \triangleright n)$  for  $n$ 
  using min-cons-hyp-def[of  $\psi \ f \triangleright n$ ] by auto
moreover have (LEAST  $i. \psi \ i \triangleright n = f \triangleright n) = \text{imin}$  if  $n \geq n_0$  for  $n$ 
proof (rule Least-equality)
  show  $\psi \ \text{imin} \triangleright n = f \triangleright n$ 
    using imin(1) by simp
  show  $\bigwedge y. \psi \ y \triangleright n = f \triangleright n \implies \text{imin} \leq y$ 
    using imin * leI that by blast
qed
ultimately have min-cons-hyp  $\psi \ (f \triangleright n) \ \downarrow = \text{imin}$  if  $n \geq n_0$  for  $n$ 
  using that by blast
then show ?thesis by auto
qed
with imin(1) show ?thesis by auto
qed
qed

```

corollary NUM-subseteq-TOTAL: $NUM \subseteq TOTAL$

```

proof
  fix U
  assume  $U \in NUM$ 
  then have  $\exists \psi \in \mathcal{R}^2. \forall f \in U. \exists i. \psi \ i = f$  by auto
  then have  $\exists \psi \in \mathcal{R}^2. U \in \text{NUM-wrt } \psi$ 
    using NUM-wrt-def by simp
  then have  $\exists \psi \ s. \text{learn-total } \psi \ U \ s$ 
    using NUM-imp-learn-total by auto
  then show  $U \in TOTAL$ 
    using TOTAL-def by auto
qed

```

The class V_0 is in $TOTAL - NUM$.

theorem NUM-subset-TOTAL: $NUM \subset TOTAL$

```

  using CP-subseteq-TOTAL FIN-not-subseteq-NUM FIN-subseteq-CP NUM-subseteq-TOTAL
  by auto

```

end

2.7 CONS is a proper subset of LIM

theory CONS-LIM

```

  imports Inductive-Inference-Basics

```

```

begin

```

That there are classes in $LIM - CONS$ was noted by Barzdin [4, 3] and Blum and Blum [5]. It was proven by Wiehagen [15] (see also Wiehagen and Zeugmann [16]). The proof uses this class:

definition U-LIMCONS :: partial1 set ($\langle U_{LIM-CONS} \rangle$) where

```

  ULIM-CONS  $\equiv \{vs \ @ \ [j] \ \odot \ p \ | \ vs \ j \ p. \ j \geq 2 \wedge p \in \mathcal{R}_{01} \wedge \varphi \ j = vs \ @ \ [j] \ \odot \ p\}$ 

```

Every function in $U_{LIM-CONS}$ carries a Gödel number greater or equal two of itself, after which only zeros and ones occur. Thus, a strategy that always outputs the rightmost value greater or equal two in the given prefix will converge to this Gödel number.

The next function searches an encoded list for the rightmost element greater or equal two.

definition $rmge2 :: partial1$ **where**

$rmge2\ e \equiv$
if $\forall i < e\text{-length}\ e. e\text{-nth}\ e\ i < 2$ *then* $Some\ 0$
else $Some\ (e\text{-nth}\ e\ (GREATEST\ i. i < e\text{-length}\ e \wedge e\text{-nth}\ e\ i \geq 2))$

lemma $rmge2$:

assumes $xs = list\text{-decode}\ e$

shows $rmge2\ e =$

(if $\forall i < length\ xs. xs\ !\ i < 2$ *then* $Some\ 0$
else $Some\ (xs\ !\ (GREATEST\ i. i < length\ xs \wedge xs\ !\ i \geq 2))$ *)*

proof –

have $(i < e\text{-length}\ e \wedge e\text{-nth}\ e\ i \geq 2) = (i < length\ xs \wedge xs\ !\ i \geq 2)$ **for** i

using $assms$ **by** $simp$

then have $(GREATEST\ i. i < e\text{-length}\ e \wedge e\text{-nth}\ e\ i \geq 2) =$

$(GREATEST\ i. i < length\ xs \wedge xs\ !\ i \geq 2)$

by $simp$

moreover have $(\forall i < length\ xs. xs\ !\ i < 2) = (\forall i < e\text{-length}\ e. e\text{-nth}\ e\ i < 2)$

using $assms$ **by** $simp$

moreover have $(GREATEST\ i. i < length\ xs \wedge xs\ !\ i \geq 2) < length\ xs$ (**is** $Greatest\ ?P < -$)

if $\neg (\forall i < length\ xs. xs\ !\ i < 2)$

using *that* $GreatestI\text{-ex-nat}$ [of $?P$] *le-less-linear order.asym* **by** $blast$

ultimately show *?thesis* **using** $rmge2\text{-def}$ $assms$ **by** $auto$

qed

lemma $rmge2\text{-init}$:

$rmge2\ (f \triangleright n) =$

(if $\forall i < Suc\ n. the\ (f\ i) < 2$ *then* $Some\ 0$
else $Some\ (the\ (f\ (GREATEST\ i. i < Suc\ n \wedge the\ (f\ i) \geq 2))$ *)*

proof –

let $?xs = prefix\ f\ n$

have $f \triangleright n = list\text{-encode}\ ?xs$ **by** $(simp\ add: init\text{-def})$

moreover have $(\forall i < Suc\ n. the\ (f\ i) < 2) = (\forall i < length\ ?xs. ?xs\ !\ i < 2)$

by $simp$

moreover have $(GREATEST\ i. i < Suc\ n \wedge the\ (f\ i) \geq 2) =$

$(GREATEST\ i. i < length\ ?xs \wedge ?xs\ !\ i \geq 2)$

using $length\text{-prefix}$ [of $f\ n$] $prefix\text{-nth}$ [of $- n\ f$] **by** $metis$

moreover have $(GREATEST\ i. i < Suc\ n \wedge the\ (f\ i) \geq 2) < Suc\ n$

if $\neg (\forall i < Suc\ n. the\ (f\ i) < 2)$

using *that* $GreatestI\text{-ex-nat}$ [of $\lambda i. i < Suc\ n \wedge the\ (f\ i) \geq 2\ n$] **by** $fastforce$

ultimately show *?thesis* **using** $rmge2$ **by** $auto$

qed

corollary $rmge2\text{-init-total}$:

assumes $total1\ f$

shows $rmge2\ (f \triangleright n) =$

(if $\forall i < Suc\ n. the\ (f\ i) < 2$ *then* $Some\ 0$
else $f\ (GREATEST\ i. i < Suc\ n \wedge the\ (f\ i) \geq 2)$ *)*

using $assms\ total1\text{-def}\ rmge2\text{-init}$ **by** $auto$

lemma $rmge2\text{-in-R1}$: $rmge2 \in \mathcal{R}$

```

proof -
  define g where
    g = Cn 3 r-ifle [r-constn 2 2, Cn 3 r-nth [Id 3 2, Id 3 0], Cn 3 r-nth [Id 3 2, Id 3 0], Id 3 1]
  then have recfn 3 g by simp
  then have g: eval g [j, r, e]  $\Downarrow$ = (if 2  $\leq$  e-nth e j then e-nth e j else r) for j r e
    using g-def by simp

  let ?h = Pr 1 Z g
  have recfn 2 ?h
    by (simp add: <recfn 3 g>)
  have h: eval ?h [j, e] =
    (if  $\forall i < j$ . e-nth e i < 2 then Some 0
     else Some (e-nth e (GREATEST i. i < j  $\wedge$  e-nth e i  $\geq$  2))) for j e
  proof (induction j)
    case 0
    then show ?case using <recfn 2 ?h> by auto
  next
    case (Suc j)
    then have eval ?h [Suc j, e] = eval g [j, the (eval ?h [j, e]), e]
      using <recfn 2 ?h> by auto
    then have *: eval ?h [Suc j, e]  $\Downarrow$ =
      (if 2  $\leq$  e-nth e j then e-nth e j
       else if  $\forall i < j$ . e-nth e i < 2 then 0
        else (e-nth e (GREATEST i. i < j  $\wedge$  e-nth e i  $\geq$  2)))
      using g Suc by auto
    show ?case
    proof (cases  $\forall i < \text{Suc } j$ . e-nth e i < 2)
      case True
      then show ?thesis using * by auto
    next
      case ex: False
      show ?thesis
      proof (cases 2  $\leq$  e-nth e j)
        case True
        then have eval ?h [Suc j, e]  $\Downarrow$ = e-nth e j
          using * by simp
        moreover have (GREATEST i. i < Suc j  $\wedge$  e-nth e i  $\geq$  2) = j
          using ex True Greatest-equality[of  $\lambda i$ . i < Suc j  $\wedge$  e-nth e i  $\geq$  2]
          by simp
        ultimately show ?thesis using ex by auto
      next
        case False
        then have  $\exists i < j$ . e-nth e i  $\geq$  2
          using ex leI less-Suc-eq by blast
        with * have eval ?h [Suc j, e]  $\Downarrow$ = e-nth e (GREATEST i. i < j  $\wedge$  e-nth e i  $\geq$  2)
          using False by (smt leD)
        moreover have (GREATEST i. i < Suc j  $\wedge$  e-nth e i  $\geq$  2) =
          (GREATEST i. i < j  $\wedge$  e-nth e i  $\geq$  2)
          using False ex by (metis less-SucI less-Suc-eq less-antisym numeral-2-eq-2)
        ultimately show ?thesis using ex by metis
      qed
    qed
  qed
  let ?hh = Cn 1 ?h [Cn 1 r-length [Id 1 0], Id 1 0]
  have recfn 1 ?hh

```

using $\langle \text{recfn } 2 \text{ ?}h \rangle$ **by** *simp*
with h **have** $hh: \text{eval ?}hh [e] \downarrow =$
 (if $\forall i < e\text{-length } e. e\text{-nth } e \ i < 2$ then 0
 else $e\text{-nth } e \ (GREATEST \ i. \ i < e\text{-length } e \wedge e\text{-nth } e \ i \geq 2)$) **for** e
by *auto*
then **have** $\text{eval ?}hh [e] = \text{rmge2 } e$ **for** e
unfolding *rmge2-def* **by** *auto*
moreover **have** *total ?}hh*
using $hh \text{ totalI1 } \langle \text{recfn } 1 \text{ ?}hh \rangle$ **by** *simp*
ultimately **show** *?thesis* **using** $\langle \text{recfn } 1 \text{ ?}hh \rangle$ **by** *blast*
qed

The first part of the main result is that $U_{LIM-CONS} \in LIM$.

lemma *U-LIMCONS-in-Lim*: $U_{LIM-CONS} \in LIM$

proof –

have $U_{LIM-CONS} \subseteq \mathcal{R}$

unfolding *U-LIMCONS-def* **using** *prepend-in-R1 RPred1-subseteq-R1* **by** *blast*

have *learn-lim* $\varphi U_{LIM-CONS} \text{ rmge2}$

proof (*rule learn-limI*)

show *environment* $\varphi U_{LIM-CONS} \text{ rmge2}$

using $\langle U-LIMCONS \subseteq \mathcal{R} \rangle$ *phi-in-P2 rmge2-def rmge2-in-R1* **by** *simp*

show $\exists i. \varphi \ i = f \wedge (\forall^\infty n. \text{rmge2 } (f \triangleright n) \downarrow = i)$ **if** $f \in U_{LIM-CONS}$ **for** f

proof –

from *that* **obtain** $vs \ j \ p$ **where**

$j: j \geq 2$

and $p: p \in \mathcal{R}_{01}$

and $s: \varphi \ j = vs \ @ \ [j] \ \odot \ p$

and $f: f = vs \ @ \ [j] \ \odot \ p$

unfolding *U-LIMCONS-def* **by** *auto*

then **have** $\varphi \ j = f$ **by** *simp*

from *that* **have** *total1 f*

using $\langle U_{LIM-CONS} \subseteq \mathcal{R} \rangle$ *R1-imp-total1 total1-def* **by** *auto*

define n_0 **where** $n_0 = \text{length } vs$

have *f-gr-n0*: $f \ n \downarrow = 0 \vee f \ n \downarrow = 1$ **if** $n > n_0$ **for** n

proof –

have $f \ n = p \ (n - n_0 - 1)$

using *that n0-def f* **by** *simp*

with *RPred1-def p* **show** *?thesis* **by** *auto*

qed

have *rmge2 (f > n) ↓ = j* **if** $n \geq n_0$ **for** n

proof –

have *n0-greatest*: $(GREATEST \ i. \ i < \text{Suc } n \wedge \text{the } (f \ i) \geq 2) = n_0$

proof (*rule Greatest-equality*)

show $n_0 < \text{Suc } n \wedge \text{the } (f \ n_0) \geq 2$

using *n0-def f that j* **by** *simp*

show $\bigwedge y. \ y < \text{Suc } n \wedge \text{the } (f \ y) \geq 2 \implies y \leq n_0$

proof –

fix y **assume** $y < \text{Suc } n \wedge 2 \leq \text{the } (f \ y)$

moreover **have** $p \in \mathcal{R} \wedge (\forall n. \ p \ n \downarrow = 0 \vee p \ n \downarrow = 1)$

using *RPred1-def p* **by** *blast*

ultimately **show** $y \leq n_0$

using *f-gr-n0*

by (*metis Suc-1 Suc-n-not-le-n Zero-neq-Suc le-less-linear le-zero-eq option.sel*)

qed

qed

have $f \ n_0 \downarrow = j$

```

    using n0-def f by simp
  then have  $\neg (\forall i < \text{Suc } n. \text{the } (f \ i) < 2)$ 
    using j that less-Suc-eq-le by auto
  then have  $\text{rmge2 } (f \triangleright n) = f \ (\text{GREATEST } i. i < \text{Suc } n \wedge \text{the } (f \ i) \geq 2)$ 
    using  $\text{rmge2-init-total } \langle \text{total1 } f \rangle$  by auto
  with  $n0\text{-greatest } \langle f \ n_0 \downarrow = j \rangle$  show ?thesis by simp
qed
with  $\langle \varphi \ j = f \rangle$  show ?thesis by auto
qed
then show ?thesis using Lim-def by auto
qed

```

The class $U_{LIM-CONS}$ is *prefix-complete*, which means that every non-empty list is the prefix of some function in $U_{LIM-CONS}$. To show this we use an auxiliary lemma: For every $f \in \mathcal{R}$ and $k \in \mathbb{N}$ the value of f at k can be replaced by a Gödel number of the function resulting from the replacement.

lemma *goedel-at*:

```

  fixes m :: nat and k :: nat
  assumes f ∈ ℛ
  shows  $\exists n \geq m. \varphi \ n = (\lambda x. \text{if } x = k \text{ then Some } n \text{ else } f \ x)$ 
proof -
  define psi :: partial1 ⇒ nat ⇒ partial2 where
    psi =  $(\lambda f \ k \ i \ x. (\text{if } x = k \text{ then Some } i \text{ else } f \ x))$ 
  have psi f k ∈ ℛ2
proof -
  obtain r where r:  $\text{recfn } 1 \ r \ \text{total } r \ \text{eval } r \ [x] = f \ x$  for x
    using assms by auto
  define r-psi where
    r-psi =  $Cn \ 2 \ r\text{-ifeq } [Id \ 2 \ 1, r\text{-dummy } 1 \ (r\text{-const } k), Id \ 2 \ 0, Cn \ 2 \ r \ [Id \ 2 \ 1]]$ 
  show ?thesis
proof (rule R2I[of r-psi])
  from r-psi-def show  $\text{recfn } 2 \ r\text{-psi}$ 
    using r(1) by simp
  have  $\text{eval } r\text{-psi } [i, x] = (\text{if } x = k \text{ then Some } i \text{ else } f \ x)$  for i x
proof -
  have  $\text{eval } (Cn \ 2 \ r \ [Id \ 2 \ 1]) [i, x] = f \ x$ 
    using r by simp
  then have  $\text{eval } r\text{-psi } [i, x] = \text{eval } r\text{-ifeq } [x, k, i, \text{the } (f \ x)]$ 
    unfolding r-psi-def using  $\langle \text{recfn } 2 \ r\text{-psi} \rangle \ r \ R1\text{-imp-total1} [OF \ \text{assms}]$ 
    by simp
  then show ?thesis using assms by simp
qed
then show  $\bigwedge x \ y. \text{eval } r\text{-psi } [x, y] = \text{psi } f \ k \ x \ y$ 
  unfolding psi-def by simp
then show total r-psi
  using totalI2[of r-psi]  $\langle \text{recfn } 2 \ r\text{-psi} \rangle \ \text{assms } \text{psi-def}$  by fastforce
qed
qed
then obtain n where  $n \geq m \ \varphi \ n = \text{psi } f \ k \ n$ 
  using assms kleene-fixed-point[of psi f k m] by auto
then show ?thesis unfolding psi-def by auto
qed

```

lemma *U-LIMCONS-prefix-complete*:

assumes $\text{length } vs > 0$
shows $\exists f \in U_{LIM-CONS}. \text{prefix } f (\text{length } vs - 1) = vs$
proof –
let $?p = \lambda -. \text{Some } 0$
let $?f = vs @ [0] \odot ?p$
have $?f \in \mathcal{R}$
using *prepend-in-R1 RPred1-subseteq-R1 const0-in-RPred1* **by** *blast*
with *goedel-at[of ?f 2 length vs]* **obtain** j **where**
 $j: j \geq 2 \ \varphi \ j = (\lambda x. \text{if } x = \text{length } vs \text{ then Some } j \text{ else } ?f \ x)$ (**is** $- = ?g$)
by *auto*
moreover **have** $g: ?g \ x = (vs @ [j] \odot ?p) \ x$ **for** x
by (*simp add: nth-append*)
ultimately **have** $?g \in U_{LIM-CONS}$
unfolding *U-LIMCONS-def* **using** *const0-in-RPred1* **by** *fastforce*
moreover **have** $\text{prefix } ?g (\text{length } vs - 1) = vs$
using *g assms prefixI prepend-associative* **by** *auto*
ultimately **show** *?thesis* **by** *auto*
qed

Roughly speaking, a strategy learning a prefix-complete class must be total because it must be defined for every prefix in the class. Technically, however, the empty list is not a prefix, and thus a strategy may diverge on input 0. We can work around this by showing that if there is a strategy learning a prefix-complete class then there is also a total strategy learning this class. We need the result only for consistent learning.

lemma *U-prefix-complete-imp-total-strategy:*

assumes $\bigwedge vs. \text{length } vs > 0 \implies \exists f \in U. \text{prefix } f (\text{length } vs - 1) = vs$
and *learn-cons* $\psi \ U \ s$

shows $\exists t. \text{total1 } t \wedge \text{learn-cons } \psi \ U \ t$

proof –

define t **where** $t = (\lambda e. \text{if } e = 0 \text{ then Some } 0 \text{ else } s \ e)$

have $s \ e \downarrow$ **if** $e > 0$ **for** e

proof –

from *that* **have** *list-decode* $e \neq []$ (**is** $?vs \neq -$)

using *list-encode-0 list-encode-decode* **by** (*metis less-imp-neq*)

then **have** $\text{length } ?vs > 0$ **by** *simp*

with *assms(1)* **obtain** f **where** $f: f \in U \ \text{prefix } f (\text{length } ?vs - 1) = ?vs$
by *auto*

with *learn-cons-def learn-limE* **have** $s \ (f \triangleright (\text{length } ?vs - 1)) \downarrow$

using *assms(2)* **by** *auto*

then **show** $s \ e \downarrow$

using $f(2)$ *init-def* **by** *auto*

qed

then **have** *total1* t

using *t-def* **by** *auto*

have $t \in \mathcal{P}$

proof –

from *assms(2)* **have** $s \in \mathcal{P}$

using *learn-consE* **by** *simp*

then **obtain** rs **where** $rs: \text{recfn } 1 \ rs \ \text{eval } rs \ [x] = s \ x$ **for** x
by *auto*

define rt **where** $rt = Cn \ 1 \ (r\text{-lifz } Z \ rs) \ [Id \ 1 \ 0, Id \ 1 \ 0]$

then **have** *recfn 1* rt

using rs **by** *auto*

moreover **have** $\text{eval } rt \ [x] = t \ x$ **for** x

using $rs \ rt\text{-def } t\text{-def}$ **by** *simp*

ultimately show *?thesis by blast*
qed
have $s (f \triangleright n) = t (f \triangleright n)$ **if** $f \in U$ **for** $f n$
unfolding *t-def by (simp add: init-neq-zero)*
then have *learn-cons $\psi U t$*
using $\langle t \in \mathcal{P} \rangle$ *assms(2) learn-consE[of $\psi U s$] learn-consI[of $\psi U t$] by simp*
with $\langle total1 t \rangle$ **show** *?thesis by auto*
qed

The proof of $U_{LIM-CONS} \notin CONS$ is by contradiction. Assume there is a consistent learning strategy S . By the previous lemma S can be assumed to be total. Moreover it outputs a consistent hypothesis for every prefix. Thus for every $e \in \mathbb{N}^+$, $S(e) \neq S(e0)$ or $S(e) \neq S(e1)$ because $S(e)$ cannot be consistent with both $e0$ and $e1$. We use this property of S to construct a function in $U_{LIM-CONS}$ for which S fails as a learning strategy. To this end we define a numbering $\psi \in \mathcal{R}^2$ with $\psi_i(0) = i$ and

$$\psi_i(x+1) = \begin{cases} 0 & \text{if } S(\psi_i^x 0) \neq S(\psi_i^x), \\ 1 & \text{otherwise.} \end{cases}$$

This numbering is recursive because S is total. The “otherwise” case is equivalent to $S(\psi_i^x 1) \neq S(\psi_i^x)$ because $S(\psi_i^x)$ cannot be consistent with both $\psi_i^x 0$ and $\psi_i^x 1$. Therefore every prefix ψ_i^x is extended in such a way that S changes its hypothesis. Hence S does not learn ψ_i in the limit. Kleene’s fixed-point theorem ensures that for some $j \geq 2$, $\varphi_j = \psi_j$. This ψ_j is the sought function in $U_{LIM-CONS}$.

The following locale formalizes the construction of ψ for a total strategy S .

locale *cons-lim =*
fixes $s :: partial1$
assumes *s-in-R1: $s \in \mathcal{R}$*
begin

A *recf* computing the strategy:

definition *r-s :: recf where*
 $r-s \equiv SOME\ r-s.\ recfn\ 1\ r-s \wedge total\ r-s \wedge s = (\lambda x.\ eval\ r-s\ [x])$

lemma *r-s-recfn [simp]: recfn 1 r-s*
and *r-s-total [simp]: $\bigwedge x.\ eval\ r-s\ [x] \downarrow$*
and *eval-r-s: $s = (\lambda x.\ eval\ r-s\ [x])$*
using *r-s-def R1-SOME[OF s-in-R1, of r-s] by simp-all*

The next function represents the prefixes of ψ_i .

fun *prefixes :: nat \Rightarrow nat \Rightarrow nat list where*
 $prefixes\ i\ 0 = [i]$
 $| prefixes\ i\ (Suc\ x) = (prefixes\ i\ x) @$
 $[if\ s\ (e-snoc\ (list-encode\ (prefixes\ i\ x))\ 0) = s\ (list-encode\ (prefixes\ i\ x))$
 $then\ 1\ else\ 0]$

definition *r-prefixes-aux \equiv*
 $Cn\ 3\ r-ifeq$
 $[Cn\ 3\ r-s\ [Cn\ 3\ r-snoc\ [Id\ 3\ 1,\ r-constn\ 2\ 0]],$
 $Cn\ 3\ r-s\ [Id\ 3\ 1],$
 $Cn\ 3\ r-snoc\ [Id\ 3\ 1,\ r-constn\ 2\ 1],$
 $Cn\ 3\ r-snoc\ [Id\ 3\ 1,\ r-constn\ 2\ 0]]$

lemma *r-prefixes-aux-recfn: recfn 3 r-prefixes-aux*

unfolding *r-prefixes-aux-def* **by** *simp*

lemma *r-prefixes-aux*:

eval r-prefixes-aux [j, v, i] ↓ =

e-snoc v (if eval r-s [e-snoc v 0] = eval r-s [v] then 1 else 0)

unfolding *r-prefixes-aux-def* **by** *auto*

definition *r-prefixes* \equiv *r-swap (Pr 1 r-singleton-encode r-prefixes-aux)*

lemma *r-prefixes-recfn*: *recfn 2 r-prefixes*

unfolding *r-prefixes-def r-prefixes-aux-def* **by** *simp*

lemma *r-prefixes*: *eval r-prefixes [i, n] ↓ = list-encode (prefixes i n)*

proof –

let *?h = Pr 1 r-singleton-encode r-prefixes-aux*

have *eval ?h [n, i] ↓ = list-encode (prefixes i n)*

proof (*induction n*)

case *0*

then show *?case*

using *r-prefixes-def r-prefixes-aux-recfn r-singleton-encode* **by** *simp*

next

case (*Suc n*)

then show *?case*

using *r-prefixes-aux-recfn r-prefixes-aux eval-r-s*

by *auto metis+*

qed

moreover have *eval ?h [n, i] = eval r-prefixes [i, n]* **for** *i n*

unfolding *r-prefixes-def* **by** (*simp add: r-prefixes-aux-recfn*)

ultimately show *?thesis* **by** *simp*

qed

lemma *prefixes-neq-nil*: *length (prefixes i x) > 0*

by (*induction x*) *auto*

The actual numbering can then be defined via *prefixes*.

definition *psi* :: *partial2* ($\langle \psi \rangle$) **where**

ψ i x \equiv *Some (last (prefixes i x))*

lemma *psi-in-R2*: $\psi \in \mathcal{R}^2$

proof

define *r-psi* **where** *r-psi* \equiv *Cn 2 r-last [r-prefixes]*

have *recfn 2 r-psi*

unfolding *r-psi-def* **by** (*simp add: r-prefixes-recfn*)

then have *eval r-psi [i, n] ↓ = last (prefixes i n)* **for** *n i*

unfolding *r-psi-def* **using** *r-prefixes r-prefixes-recfn prefixes-neq-nil* **by** *simp*

then have ($\lambda i x. \text{Some (last (prefixes i x))}$) $\in \mathcal{P}^2$

using $\langle \text{recfn } 2 \text{ r-psi} \rangle$ *P2I[of r-psi]* **by** *simp*

with *psi-def* **show** $\psi \in \mathcal{P}^2$ **by** *presburger*

moreover show *total2 psi*

unfolding *psi-def* **by** *auto*

qed

lemma *psi-0-or-1*:

assumes $n > 0$

shows $\psi i n \downarrow = 0 \vee \psi i n \downarrow = 1$

proof –

from *assms* **obtain** m **where** $n = \text{Suc } m$
using *gr0-implies-Suc* **by** *blast*
then have $\text{last } (\text{prefixes } i \text{ (Suc } m)) = 0 \vee \text{last } (\text{prefixes } i \text{ (Suc } m)) = 1$
by *simp*
then show *?thesis* **using** $\langle n = \text{Suc } m \rangle$ *psi-def* **by** *simp*
qed

The function *prefixes* does indeed provide the prefixes for ψ .

lemma *psi-init*: $(\psi \ i) \triangleright x = \text{list-encode } (\text{prefixes } i \ x)$
proof –
have $\text{prefix } (\psi \ i) \ x = \text{prefixes } i \ x$
unfolding *psi-def*
by (*induction* x) (*simp-all* *add*: *prefix-0* *prefix-Suc*)
with *init-def* **show** *?thesis* **by** *simp*
qed

One of the functions ψ_i is in $U_{LIM-CONS}$.

lemma *ex-psi-in-U*: $\exists j. \psi \ j \in U_{LIM-CONS}$
proof –
obtain j **where** $j \geq 2$ $\psi \ j = \varphi \ j$
using *kleene-fixed-point*[*of* ψ] *psi-in-R2* *R2-imp-P2* **by** *metis*
then have $\psi \ j \in \mathcal{P}$ **by** (*simp* *add*: *phi-in-P2*)
define p **where** $p = (\lambda x. \psi \ j \ (x + 1))$
have $p \in \mathcal{R}_{01}$
proof –
from *p-def* $\langle \psi \ j \in \mathcal{P} \rangle$ *skip-P1* **have** $p \in \mathcal{P}$ **by** *blast*
from *psi-in-R2* **have** *total1* $(\psi \ j)$ **by** *simp*
with *p-def* **have** *total1* p
by (*simp* *add*: *total1-def*)
with *psi-0-or-1* **have** $p \ n \downarrow = 0 \vee p \ n \downarrow = 1$ **for** n
using *psi-def* *p-def* **by** *simp*
then show *?thesis*
by (*simp* *add*: *RPred1-def* *P1-total-imp-R1* $\langle p \in \mathcal{P} \rangle$ $\langle \text{total1 } p \rangle$)
qed
moreover have $\psi \ j = [j] \odot p$
proof
fix x
show $\psi \ j \ x = ([j] \odot p) \ x$
proof (*cases* $x = 0$)
case *True*
then show *?thesis* **using** *psi-def* *psi-def* *prepend-at-less* **by** *simp*
next
case *False*
then show *?thesis* **using** *p-def* **by** *simp*
qed
qed
ultimately have $\psi \ j \in U_{LIM-CONS}$
using j *U-LIMCONS-def* **by** (*metis* (*mono-tags*, *lifting*) *append-Nil* *mem-Collect-eq*)
then show *?thesis* **by** *auto*
qed

The strategy fails to learn $U_{LIM-CONS}$ because it changes its hypothesis all the time on functions $\psi_j \in V_0$.

lemma *U-LIMCONS-not-learn-cons*: $\neg \text{learn-cons } \varphi \ U_{LIM-CONS} \ s$
proof

assume $learn: learn-cons \varphi U_{LIM-CONS} s$
have $s (list-encode (vs @ [0])) \neq s (list-encode (vs @ [1]))$ **for** vs
proof –
obtain f_0 **where** $f_0: f_0 \in U_{LIM-CONS} prefix f_0 (length vs) = vs @ [0]$
using $U-LIMCONS-prefix-complete[of vs @ [0]]$ **by** $auto$
obtain f_1 **where** $f_1: f_1 \in U_{LIM-CONS} prefix f_1 (length vs) = vs @ [1]$
using $U-LIMCONS-prefix-complete[of vs @ [1]]$ **by** $auto$
have $f_0 (length vs) \neq f_1 (length vs)$
using $f_0 f_1$ **by** $(metis lessI nth-append-length prefix-nth zero-neq-one)$
moreover have $\varphi (the (s (f_0 \triangleright length vs))) (length vs) = f_0 (length vs)$
using $learn-consE(3)[of \varphi U-LIMCONS s, OF learn, of f_0 length vs, OF f_0(1)]$
by $simp$
moreover have $\varphi (the (s (f_1 \triangleright length vs))) (length vs) = f_1 (length vs)$
using $learn-consE(3)[of \varphi U-LIMCONS s, OF learn, of f_1 length vs, OF f_1(1)]$
by $simp$
ultimately have $the (s (f_0 \triangleright length vs)) \neq the (s (f_1 \triangleright length vs))$
by $auto$
then have $s (f_0 \triangleright length vs) \neq s (f_1 \triangleright length vs)$
by $auto$
with $f_0(2) f_1(2)$ **show** $?thesis$ **by** $(simp add: init-def)$
qed
then have $s (list-encode (vs @ [0])) \neq s (list-encode vs) \vee$
 $s (list-encode (vs @ [1])) \neq s (list-encode vs)$
for vs
by $metis$
then have $s (list-encode (prefixes i (Suc x))) \neq s (list-encode (prefixes i x))$ **for** $i x$
by $simp$
then have $\neg learn-lim \varphi \{\psi i\} s$ **for** i
using $psi-def psi-init always-hyp-change-not-Lim$ **by** $simp$
then have $\neg learn-lim \varphi U-LIMCONS s$
using $ex-psi-in-U learn-lim-closed-subseteq$ **by** $blast$
then show $False$
using $learn learn-cons-def$ **by** $simp$
qed
end

With the locale we can now show the second part of the main result:

lemma $U-LIMCONS-not-in-CONS: U_{LIM-CONS} \notin CONS$

proof

assume $U_{LIM-CONS} \in CONS$
then have $U_{LIM-CONS} \in CONS-wrt \varphi$
by $(simp add: CONS-wrt-phi-eq-CONS)$
then obtain $almost-s$ **where** $learn-cons \varphi U_{LIM-CONS} almost-s$
using $CONS-wrt-def$ **by** $auto$
then obtain s **where** $s: total1 s learn-cons \varphi U_{LIM-CONS} s$
using $U-LIMCONS-prefix-complete U-prefix-complete-imp-total-strategy$ **by** $blast$
then have $s \in \mathcal{R}$
using $learn-consE(1) P1-total-imp-R1$ **by** $blast$
with $cons-lim-def$ **interpret** $cons-lim s$ **by** $simp$
show $False$
using $s(2) U-LIMCONS-not-learn-cons$ **by** $simp$
qed

The main result of this section:

theorem $CONS-subset-Lim: CONS \subset LIM$

using *U-LIMCONS-in-Lim U-LIMCONS-not-in-CONS CONS-subseteq-Lim* by *auto*

end

2.8 Lemma R

theory *Lemma-R*

imports *Inductive-Inference-Basics*

begin

A common technique for constructing a class that cannot be learned is diagonalization against all strategies (see, for instance, Section 2.9). Similarly, the typical way of proving that a class cannot be learned is by assuming there is a strategy and deriving a contradiction. Both techniques are easier to carry out if one has to consider only *total* recursive strategies. This is not possible in general, since after all the definitions of the inference types admit strictly partial strategies. However, for many inference types one can show that for every strategy there is a total strategy with at least the same “learning power”. Results to that effect are called Lemma R.

Lemma R comes in different strengths depending on how general the construction of the total recursive strategy is. CONS is the only inference type considered here for which not even a weak form of Lemma R holds.

2.8.1 Strong Lemma R for LIM, FIN, and BC

In its strong form Lemma R says that for any strategy S , there is a total strategy T that learns all classes S learns regardless of hypothesis space. The strategy T can be derived from S by a delayed simulation of S . More precisely, for input f^n , T simulates S for prefixes f^0, f^1, \dots, f^n for at most n steps. If S halts on none of the prefixes, T outputs an arbitrary hypothesis. Otherwise let $k \leq n$ be maximal such that S halts on f^k in at most n steps. Then T outputs $S(f^k)$.

We reformulate some lemmas for *r-result1* to make it easier to use them with φ .

lemma *r-result1-converg-phi*:

assumes $\varphi \ i \ x \ \downarrow = \ v$

shows $\exists t.$

$(\forall t' \geq t. \text{eval } r\text{-result1 } [t', i, x] \downarrow = \text{Suc } v) \wedge$

$(\forall t' < t. \text{eval } r\text{-result1 } [t', i, x] \downarrow = 0)$

using *assms r-result1-converg' phi-def* by *simp-all*

lemma *r-result1-bivalent'*:

assumes $\text{eval } r\text{-phi } [i, x] \downarrow = \ v$

shows $\text{eval } r\text{-result1 } [t, i, x] \downarrow = \ \text{Suc } v \vee \text{eval } r\text{-result1 } [t, i, x] \downarrow = \ 0$

using *assms r-result1 r-result-bivalent' r-phi''* by *simp*

lemma *r-result1-bivalent-phi*:

assumes $\varphi \ i \ x \ \downarrow = \ v$

shows $\text{eval } r\text{-result1 } [t, i, x] \downarrow = \ \text{Suc } v \vee \text{eval } r\text{-result1 } [t, i, x] \downarrow = \ 0$

using *assms r-result1-bivalent' phi-def* by *simp-all*

lemma *r-result1-diverg-phi*:

assumes $\varphi \ i \ x \ \uparrow$

shows $\text{eval } r\text{-result1 } [t, i, x] \downarrow = \ 0$

using *assms phi-def r-result1-diverg'* by *simp*

lemma *r-result1-some-phi*:
assumes *eval r-result1 [t, i, x] ↓= Suc v*
shows $\varphi\ i\ x\ ↓= v$
using *assms phi-def r-result1-Some'* by *simp*

lemma *r-result1-saturating'*:
assumes *eval r-result1 [t, i, x] ↓= Suc v*
shows *eval r-result1 [t + d, i, x] ↓= Suc v*
using *assms r-result1 r-result-saturating r-phi''* by *simp*

lemma *r-result1-saturating-the*:
assumes *the (eval r-result1 [t, i, x]) > 0 and t' ≥ t*
shows *the (eval r-result1 [t', i, x]) > 0*
proof –
from *assms(1)* **obtain** *v* **where** *eval r-result1 [t, i, x] ↓= Suc v*
using *r-result1-bivalent-phi r-result1-diverg-phi*
by (*metis inc-induct le-0-eq not-less-zero option.discI option.expand option.sel*)
with *assms* **have** *eval r-result1 [t', i, x] ↓= Suc v*
using *r-result1-saturating' le-Suc-ex* **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *Greatest-bounded-Suc*:
fixes *P :: nat ⇒ nat*
shows (*if P n > 0 then Suc n*
else if ∃ j < n. P j > 0 then Suc (GREATEST j. j < n ∧ P j > 0) else 0) =
(if ∃ j < Suc n. P j > 0 then Suc (GREATEST j. j < Suc n ∧ P j > 0) else 0)
(is ?lhs = ?rhs)
proof (*cases ∃ j < Suc n. P j > 0*)
case *1: True*
show *?thesis*
proof (*cases P n > 0*)
case *True*
then have (*GREATEST j. j < Suc n ∧ P j > 0*) = *n*
using *Greatest-equality[of λj. j < Suc n ∧ P j > 0]* **by** *simp*
moreover have *?rhs = Suc (GREATEST j. j < Suc n ∧ P j > 0)*
using *1* **by** *simp*
ultimately have *?rhs = Suc n* **by** *simp*
then show *?thesis* **using** *True* **by** *simp*
next
case *False*
then have *?lhs = Suc (GREATEST j. j < n ∧ P j > 0)*
using *1* **by** (*metis less-SucE*)
moreover have *?rhs = Suc (GREATEST j. j < Suc n ∧ P j > 0)*
using *1* **by** *simp*
moreover have (*GREATEST j. j < n ∧ P j > 0*) =
(GREATEST j. j < Suc n ∧ P j > 0)
using *1 False* **by** (*metis less-SucI less-Suc-eq*)
ultimately show *?thesis* **by** *simp*
qed
next
case *False*
then show *?thesis* **by** *auto*
qed

For n, i, x , the next function simulates φ_i on all non-empty prefixes of at most length n of the list x for at most n steps. It returns the length of the longest such prefix for which φ_i halts, or zero if φ_i does not halt for any prefix.

definition $r\text{-delay-aux} \equiv$

```
Pr 2 (r-constn 1 0)
(Cn 4 r-ifz
 [Cn 4 r-result1
  [Cn 4 r-length [Id 4 3], Id 4 2,
   Cn 4 r-take [Cn 4 S [Id 4 0], Id 4 3]],
 Id 4 1, Cn 4 S [Id 4 0]])
```

lemma $r\text{-delay-aux-prim}$: $\text{prim-recfn } 3 \text{ } r\text{-delay-aux}$

unfolding $r\text{-delay-aux-def}$ **by** simp-all

lemma $r\text{-delay-aux-total}$: $\text{total } r\text{-delay-aux}$

using $\text{prim-recfn-total}[OF \text{ } r\text{-delay-aux-prim}]$.

lemma $r\text{-delay-aux}$:

assumes $n \leq e\text{-length } x$

shows $\text{eval } r\text{-delay-aux } [n, i, x] \downarrow =$

(if $\exists j < n$. the (eval $r\text{-result1 } [e\text{-length } x, i, e\text{-take } (Suc \ j) \ x]$) > 0
then $Suc \ (GREATEST \ j$.

$j < n \wedge$

the (eval $r\text{-result1 } [e\text{-length } x, i, e\text{-take } (Suc \ j) \ x]$) > 0)

else 0)

proof –

define z **where** $z \equiv$

$Cn \ 4 \ r\text{-result1}$

$[Cn \ 4 \ r\text{-length } [Id \ 4 \ 3], Id \ 4 \ 2, Cn \ 4 \ r\text{-take } [Cn \ 4 \ S \ [Id \ 4 \ 0], Id \ 4 \ 3]]$

then have $z\text{-recfn}$: $\text{recfn } 4 \ z$ **by** simp

have z : $\text{eval } z \ [j, r, i, x] = \text{eval } r\text{-result1 } [e\text{-length } x, i, e\text{-take } (Suc \ j) \ x]$

if $j < e\text{-length } x$ **for** $j \ r \ i \ x$

unfolding $z\text{-def}$ **using** that **by** simp

define g **where** $g \equiv Cn \ 4 \ r\text{-ifz } [z, Id \ 4 \ 1, Cn \ 4 \ S \ [Id \ 4 \ 0]]$

then have g : $\text{eval } g \ [j, r, i, x] \downarrow =$

(if the (eval $r\text{-result1 } [e\text{-length } x, i, e\text{-take } (Suc \ j) \ x]$) > 0 then $Suc \ j$ else r)

if $j < e\text{-length } x$ **for** $j \ r \ i \ x$

using that $z \ \text{prim-recfn-total } z\text{-recfn}$ **by** simp

show $?thesis$

using assms

proof (induction n)

case 0

moreover have $\text{eval } r\text{-delay-aux } [0, i, x] \downarrow = 0$

using $\text{eval-Pr-0 } r\text{-delay-aux-def } r\text{-delay-aux-prim } r\text{-constn}$

by (simp add: $r\text{-delay-aux-def}$)

ultimately show $?case$ **by** simp

next

case (Suc n)

let $?P = \lambda j$. the (eval $r\text{-result1 } [e\text{-length } x, i, e\text{-take } (Suc \ j) \ x]$)

have $\text{eval } r\text{-delay-aux } [n, i, x] \downarrow$

using Suc **by** simp

moreover have $\text{eval } r\text{-delay-aux } [Suc \ n, i, x] =$

$\text{eval } (Pr \ 2 \ (r\text{-constn } 1 \ 0) \ g) \ [Suc \ n, i, x]$

unfolding *r-delay-aux-def g-def z-def* **by** *simp*
ultimately have *eval r-delay-aux [Suc n, i, x] =*
eval g [n, the (eval r-delay-aux [n, i, x]), i, x]
using *r-delay-aux-prim Suc eval-Pr-converg-Suc*
by (*simp add: r-delay-aux-def g-def z-def numeral-3-eq-3*)
then have *eval r-delay-aux [Suc n, i, x] ↓=*
(if ?P n > 0 then Suc n
else if ∃ j < n. ?P j > 0 then Suc (GREATEST j. j < n ∧ ?P j > 0) else 0)
using *g Suc* **by** *simp*
then have *eval r-delay-aux [Suc n, i, x] ↓=*
(if ∃ j < Suc n. ?P j > 0 then Suc (GREATEST j. j < Suc n ∧ ?P j > 0) else 0)
using *Greatest-bounded-Suc[where ?P=?P]* **by** *simp*
then show *?case* **by** *simp*
qed
qed

The next function simulates φ_i on all non-empty prefixes of a list x of length n for at most n steps and outputs the length of the longest prefix for which φ_i halts, or zero if φ_i does not halt for any such prefix.

definition *r-delay* \equiv *Cn 2 r-delay-aux [Cn 2 r-length [Id 2 1], Id 2 0, Id 2 1]*

lemma *r-delay-recfn [simp]: recfn 2 r-delay*
unfolding *r-delay-def* **by** (*simp add: r-delay-aux-prim*)

lemma *r-delay:*
eval r-delay [i, x] ↓=
(if ∃ j < e-length x. the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0
then Suc (GREATEST j.
j < e-length x ∧ the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0)
else 0)
unfolding *r-delay-def* **using** *r-delay-aux r-delay-aux-prim* **by** *simp*

definition *delay i x* \equiv *Some*
(if ∃ j < e-length x. the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0
then Suc (GREATEST j.
j < e-length x ∧ the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0)
else 0)

lemma *delay-in-R2: delay ∈ ℝ²*
using *r-delay totalI2 R2I delay-def r-delay-recfn*
by (*metis (no-types, lifting) numeral-2-eq-2 option.simps(3)*)

lemma *delay-le-length: the (delay i x) ≤ e-length x*

proof (*cases ∃ j < e-length x. the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0*)

case *True*

let *?P = λj. j < e-length x ∧ the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0*

from *True* **have** $\exists j. ?P j$ **by** *simp*

moreover have $\bigwedge y. ?P y \implies y \leq e\text{-length } x$ **by** *simp*

ultimately have $?P (\text{Greatest } ?P)$

using *GreatestI-ex-nat[where ?P=?P]* **by** *blast*

then have $\text{Greatest } ?P < e\text{-length } x$ **by** *simp*

moreover have $\text{delay } i \ x \downarrow = \text{Suc } (\text{Greatest } ?P)$

using *delay-def True* **by** *simp*

ultimately show *?thesis* **by** *auto*

next

case *False*

then show *?thesis* using *delay-def* by *auto*
qed

lemma *e-take-delay-init*:

assumes $f \in \mathcal{R}$ and the $(\text{delay } i (f \triangleright n)) > 0$
shows $e\text{-take } (\text{the } (\text{delay } i (f \triangleright n))) (f \triangleright n) = f \triangleright (\text{the } (\text{delay } i (f \triangleright n)) - 1)$
using *assms e-take-init[of f - n]* *length-init[of f n]* *delay-le-length[of i f \triangleright n]*
by (*metis One-nat-def Suc-le-lessD Suc-pred*)

lemma *delay-gr0-converg*:

assumes the $(\text{delay } i x) > 0$
shows $\varphi i (e\text{-take } (\text{the } (\text{delay } i x)) x) \downarrow$

proof –

let $?P = \lambda j. j < e\text{-length } x \wedge \text{the } (\text{eval } r\text{-result1 } [e\text{-length } x, i, e\text{-take } (\text{Suc } j) x]) > 0$

have $\exists j. ?P j$

proof (*rule ccontr*)

assume $\neg (\exists j. ?P j)$

then have $\text{delay } i x \downarrow = 0$

using *delay-def* by *simp*

with *assms* show *False* by *simp*

qed

then have $d: \text{the } (\text{delay } i x) = \text{Suc } (\text{Greatest } ?P)$

using *delay-def* by *simp*

moreover have $\bigwedge y. ?P y \implies y \leq e\text{-length } x$ by *simp*

ultimately have $?P (\text{Greatest } ?P)$

using $\langle \exists j. ?P j \rangle$ *GreatestI-ex-nat*[**where** $?P=?P$] by *blast*

then have the $(\text{eval } r\text{-result1 } [e\text{-length } x, i, e\text{-take } (\text{Suc } (\text{Greatest } ?P)) x]) > 0$

by *simp*

then have the $(\text{eval } r\text{-result1 } [e\text{-length } x, i, e\text{-take } (\text{the } (\text{delay } i x)) x]) > 0$

using d by *simp*

then show *?thesis* using *r-result1-diverg-phi* by *fastforce*

qed

lemma *delay-unbounded*:

fixes $n :: \text{nat}$

assumes $f \in \mathcal{R}$ and $\forall n. \varphi i (f \triangleright n) \downarrow$

shows $\exists m. \text{the } (\text{delay } i (f \triangleright m)) > n$

proof –

from *assms* have $\exists t. \text{the } (\text{eval } r\text{-result1 } [t, i, f \triangleright n]) > 0$

using *r-result1-converg-phi*

by (*metis le-refl option.exhaust-sel option.sel zero-less-Suc*)

then obtain t where $t: \text{the } (\text{eval } r\text{-result1 } [t, i, f \triangleright n]) > 0$

by *auto*

let $?m = \max n t$

have $\text{Suc } ?m \geq t$ by *simp*

have $m: \text{the } (\text{eval } r\text{-result1 } [\text{Suc } ?m, i, f \triangleright n]) > 0$

proof –

let $?w = \text{eval } r\text{-result1 } [t, i, f \triangleright n]$

obtain v where $v: ?w \downarrow = \text{Suc } v$

using t *assms*(2) *r-result1-bivalent-phi* by *fastforce*

have $\text{eval } r\text{-result1 } [\text{Suc } ?m, i, f \triangleright n] = ?w$

using v t *r-result1-saturating'* $\langle \text{Suc } ?m \geq t \rangle$ *le-Suc-ex* by *fastforce*

then show *?thesis* using t by *simp*

qed

let $?x = f \triangleright ?m$

have the $(\text{delay } i ?x) > n$

proof –
 let $?P = \lambda j. j < e\text{-length } ?x \wedge \text{the } (eval\ r\text{-result1 } [e\text{-length } ?x, i, e\text{-take } (Suc\ j) ?x]) > 0$
 have $e\text{-length } ?x = Suc\ ?m$ **by** *simp*
 moreover have $e\text{-take } (Suc\ n) ?x = f \triangleright n$
 using *assms(1) e-take-init* **by** *auto*
 ultimately have $?P\ n$
 using *m* **by** *simp*
 have $\bigwedge y. ?P\ y \implies y \leq e\text{-length } ?x$ **by** *simp*
 with $\langle ?P\ n \rangle$ have $n \leq (Greatest\ ?P)$
 using *Greatest-le-nat[of ?P n e-length ?x]* **by** *simp*
 moreover have $\text{the } (delay\ i\ ?x) = Suc\ (Greatest\ ?P)$
 using *delay-def* $\langle ?P\ n \rangle$ **by** *auto*
 ultimately show *?thesis* **by** *simp*
qed
 then show *?thesis* **by** *auto*
qed

lemma *delay-monotone*:

assumes $f \in \mathcal{R}$ **and** $n_1 \leq n_2$
 shows $\text{the } (delay\ i\ (f \triangleright n_1)) \leq \text{the } (delay\ i\ (f \triangleright n_2))$
 (is $\text{the } (delay\ i\ ?x_1) \leq \text{the } (delay\ i\ ?x_2)$)
proof (cases $\text{the } (delay\ i\ (f \triangleright n_1)) = 0$)
 case *True*
 then show *?thesis* **by** *simp*
next
 case *False*
 let $?P1 = \lambda j. j < e\text{-length } ?x_1 \wedge \text{the } (eval\ r\text{-result1 } [e\text{-length } ?x_1, i, e\text{-take } (Suc\ j) ?x_1]) > 0$
 let $?P2 = \lambda j. j < e\text{-length } ?x_2 \wedge \text{the } (eval\ r\text{-result1 } [e\text{-length } ?x_2, i, e\text{-take } (Suc\ j) ?x_2]) > 0$
 from *False* have *d1*: $\text{the } (delay\ i\ ?x_1) = Suc\ (Greatest\ ?P1) \exists j. ?P1\ j$
 using *delay-def option.collapse* **by** *fastforce+*
 moreover have $\bigwedge y. ?P1\ y \implies y \leq e\text{-length } ?x_1$ **by** *simp*
 ultimately have $*$: $?P1\ (Greatest\ ?P1)$ **using** *GreatestI-ex-nat[of ?P1]* **by** *blast*
 let $?j = Greatest\ ?P1$
 from $*$ have $?j < e\text{-length } ?x_1$ **by** *auto*
 then have *1*: $e\text{-take } (Suc\ ?j) ?x_1 = e\text{-take } (Suc\ ?j) ?x_2$
 using *assms e-take-init* **by** *auto*
 from $*$ have *2*: $?j < e\text{-length } ?x_2$ **using** *assms(2)* **by** *auto*
 with *1* $*$ have $\text{the } (eval\ r\text{-result1 } [e\text{-length } ?x_1, i, e\text{-take } (Suc\ ?j) ?x_2]) > 0$
 by *simp*
 moreover have $e\text{-length } ?x_1 \leq e\text{-length } ?x_2$
 using *assms(2)* **by** *auto*
 ultimately have $\text{the } (eval\ r\text{-result1 } [e\text{-length } ?x_2, i, e\text{-take } (Suc\ ?j) ?x_2]) > 0$
 using *r-result1-saturating-the* **by** *simp*
 with *2* have $?P2\ ?j$ **by** *simp*
 then have *d2*: $\text{the } (delay\ i\ ?x_2) = Suc\ (Greatest\ ?P2)$
 using *delay-def* **by** *auto*
 have $\bigwedge y. ?P2\ y \implies y \leq e\text{-length } ?x_2$ **by** *simp*
 with $\langle ?P2\ ?j \rangle$ have $?j \leq (Greatest\ ?P2)$ **using** *Greatest-le-nat[of ?P2]* **by** *blast*
 with *d1 d2* show *?thesis* **by** *simp*
qed

lemma *delay-unbounded-monotone*:

fixes $n :: nat$
assumes $f \in \mathcal{R}$ **and** $\forall n. \varphi\ i\ (f \triangleright n) \downarrow$
 shows $\exists m_0. \forall m \geq m_0. \text{the } (delay\ i\ (f \triangleright m)) > n$
proof –

from *assms* *delay-unbounded* **obtain** m_0 **where** *the* (*delay* i ($f \triangleright m_0$)) $> n$
by *blast*
then have $\forall m \geq m_0. \text{the } (\text{delay } i \text{ } (f \triangleright m)) > n$
using *assms*(1) *delay-monotone order.strict-trans2* **by** *blast*
then show *?thesis* **by** *auto*
qed

Now we can define a function that simulates an arbitrary strategy φ_i in a delayed way. The parameter d is the default hypothesis for when φ_i does not halt within the time bound for any prefix.

definition *r-totalizer* $:: \text{nat} \Rightarrow \text{recf}$ **where**

r-totalizer $d \equiv$
Cn 2
(*r-lifz*
(*r-constn* 1 d)
(*Cn* 2 *r-phi*
[*Id* 2 0, *Cn* 2 *r-take* [*Cn* 2 *r-delay* [*Id* 2 0, *Id* 2 1], *Id* 2 1]])
[*Cn* 2 *r-delay* [*Id* 2 0, *Id* 2 1], *Id* 2 0, *Id* 2 1])

lemma *r-totalizer-recfn*: *recfn* 2 (*r-totalizer* d)
unfolding *r-totalizer-def* **by** *simp*

lemma *r-totalizer*:

eval (*r-totalizer* d) [i , x] =
(*if* *the* (*delay* i x) = 0 *then* *Some* d *else* φ i (*e-take* (*the* (*delay* i x)) x))

proof –

let $?i = \text{Cn } 2 \text{ } r\text{-delay } [Id \ 2 \ 0, Id \ 2 \ 1]$
have *eval* $?i$ [i , x] = *eval* *r-delay* [i , x] **for** i x
using *r-delay-recfn* **by** *simp*
then have i : *eval* $?i$ [i , x] = *delay* i x **for** i x
using *r-delay* **by** (*simp* *add*: *delay-def*)
let $?t = \text{r-constn } 1 \ d$
have t : *eval* $?t$ [i , x] \Downarrow d **for** i x **by** *simp*
let $?e1 = \text{Cn } 2 \text{ } r\text{-take } [?i, Id \ 2 \ 1]$
let $?e = \text{Cn } 2 \text{ } r\text{-phi } [Id \ 2 \ 0, ?e1]$
have *eval* $?e1$ [i , x] = *eval* *r-take* [*the* (*delay* i x), x] **for** i x
using *r-delay* i *delay-def* **by** *simp*
then have *eval* $?e1$ [i , x] \Downarrow *e-take* (*the* (*delay* i x)) x **for** i x
using *delay-le-length* **by** *simp*
then have e : *eval* $?e$ [i , x] = φ i (*e-take* (*the* (*delay* i x)) x)
using *phi-def* **by** *simp*
let $?z = \text{r-lifz } ?t \ ?e$
have *recfn-te*: *recfn* 2 $?t$ *recfn* 2 $?e$
by *simp-all*
then have *eval* (*r-totalizer* d) [i , x] = *eval* (*r-lifz* $?t$ $?e$) [*the* (*delay* i x), i , x]
for i x
unfolding *r-totalizer-def* **using** i *r-totalizer-recfn* *delay-def* **by** *simp*
then have *eval* (*r-totalizer* d) [i , x] =
(*if* *the* (*delay* i x) = 0 *then* *eval* $?t$ [i , x] *else* *eval* $?e$ [i , x])
for i x
using *recfn-te* **by** *simp*
then show *?thesis* **using** t e **by** *simp*

qed

lemma *r-totalizer-total*: *total* (*r-totalizer* d)

proof (*rule totalI2*)

show *recfn 2 (r-totalizer d) using r-totalizer-recfn by simp*
show $\bigwedge x y. \text{eval } (r\text{-totalizer } d) [x, y] \downarrow$
using *r-totalizer delay-gr0-converg by simp*
qed

definition *totalizer :: nat \Rightarrow partial2 where*
totalizer d i x \equiv
if the (delay i x) = 0 then Some d else φ i (e-take (the (delay i x)) x)

lemma *totalizer-init:*
assumes $f \in \mathcal{R}$
shows *totalizer d i (f \triangleright n) =*
(if the (delay i (f \triangleright n)) = 0 then Some d
else φ i (f \triangleright (the (delay i (f \triangleright n)) - 1)))
using *assms e-take-delay-init by (simp add: totalizer-def)*

lemma *totalizer-in-R2: totalizer d $\in \mathcal{R}^2$*
using *totalizer-def r-totalizer r-totalizer-total R2I r-totalizer-recfn*
by *metis*

For LIM, *totalizer* works with every default hypothesis *d*.

lemma *lemma-R-for-Lim:*
assumes *learn-lim ψ U (φ i)*
shows *learn-lim ψ U (totalizer d i)*
proof (*rule learn-limI*)
show *env: environment ψ U (totalizer d i)*
using *assms learn-limE(1) totalizer-in-R2 by auto*
show $\exists j. \psi j = f \wedge (\forall^\infty n. \text{totalizer } d \ i \ (f \triangleright n) \downarrow = j)$ **if** $f \in U$ **for** f
proof –
have $f \in \mathcal{R}$
using *assms env that by auto*
from *assms learn-limE obtain j n₀ where*
j: $\psi j = f$ and
n0: $\forall n \geq n_0. (\varphi i) (f \triangleright n) \downarrow = j$
using $\langle f \in U \rangle$ **by** *metis*
obtain m_0 **where** $m_0: \forall m \geq m_0. \text{the } (delay \ i \ (f \triangleright m)) > n_0$
using *delay-unbounded-monotone $\langle f \in \mathcal{R} \rangle \langle f \in U \rangle$ assms learn-limE(1)*
by *blast*
then **have** $\forall m \geq m_0. \text{totalizer } d \ i \ (f \triangleright m) = \varphi \ i \ (e\text{-take } (the \ (delay \ i \ (f \triangleright m))) \ (f \triangleright m))$
using *totalizer-def by auto*
then **have** $\forall m \geq m_0. \text{totalizer } d \ i \ (f \triangleright m) = \varphi \ i \ (f \triangleright (the \ (delay \ i \ (f \triangleright m)) - 1))$
using *e-take-delay-init m0 $\langle f \in \mathcal{R} \rangle$ by auto*
with $m_0 \ n_0$ **have** $\forall m \geq m_0. \text{totalizer } d \ i \ (f \triangleright m) \downarrow = j$
by *auto*
with j **show** *?thesis by auto*
qed
qed

The effective version of Lemma R for LIM states that there is a total recursive function computing Gödel numbers of total strategies from those of arbitrary strategies.

lemma *lemma-R-for-Lim-effective:*
 $\exists g \in \mathcal{R}. \forall i.$
 $\varphi (the (g \ i)) \in \mathcal{R} \wedge$
 $(\forall U \psi. \text{learn-lim } \psi \ U \ (\varphi \ i) \longrightarrow \text{learn-lim } \psi \ U \ (\varphi (the (g \ i))))$
proof –

have *totalizer* $0 \in \mathcal{P}^2$ **using** *totalizer-in-R2* **by** *auto*
then obtain g **where** $g: g \in \mathcal{R} \forall i. (\text{totalizer } 0) i = \varphi (\text{the } (g i))$
using *numbering-translation-for-phi* **by** *blast*
with *totalizer-in-R2* **have** $\forall i. \varphi (\text{the } (g i)) \in \mathcal{R}$
by (*metis R2-proj-R1*)
moreover from $g(2)$ *lemma-R-for-Lim*[**where** $?d=0$] **have**
 $\forall i U \psi. \text{learn-lim } \psi U (\varphi i) \longrightarrow \text{learn-lim } \psi U (\varphi (\text{the } (g i)))$
by *simp*
ultimately show $?thesis$ **using** $g(1)$ **by** *blast*
qed

In order for us to use the previous lemma, we need a function that performs the actual computation:

definition *r-limr* \equiv
SOME $g.$
 $\text{recfn } 1 g \wedge$
 $\text{total } g \wedge$
 $(\forall i. \varphi (\text{the } (\text{eval } g [i])) \in \mathcal{R} \wedge$
 $(\forall U \psi. \text{learn-lim } \psi U (\varphi i) \longrightarrow \text{learn-lim } \psi U (\varphi (\text{the } (\text{eval } g [i])))))$

lemma *r-limr-recfn*: $\text{recfn } 1 r\text{-limr}$
and *r-limr-total*: $\text{total } r\text{-limr}$
and *r-limr*:
 $\varphi (\text{the } (\text{eval } r\text{-limr } [i])) \in \mathcal{R}$
 $\text{learn-lim } \psi U (\varphi i) \Longrightarrow \text{learn-lim } \psi U (\varphi (\text{the } (\text{eval } r\text{-limr } [i])))$

proof –
let $?P = \lambda g.$
 $g \in \mathcal{R} \wedge$
 $(\forall i. \varphi (\text{the } (g i)) \in \mathcal{R} \wedge (\forall U \psi. \text{learn-lim } \psi U (\varphi i) \longrightarrow \text{learn-lim } \psi U (\varphi (\text{the } (g i)))))$
let $?Q = \lambda g.$
 $\text{recfn } 1 g \wedge$
 $\text{total } g \wedge$
 $(\forall i. \varphi (\text{the } (\text{eval } g [i])) \in \mathcal{R} \wedge$
 $(\forall U \psi. \text{learn-lim } \psi U (\varphi i) \longrightarrow \text{learn-lim } \psi U (\varphi (\text{the } (\text{eval } g [i])))))$
have $\exists g. ?P g$ **using** *lemma-R-for-Lim-effective* **by** *auto*
then obtain g **where** $?P g$ **by** *auto*
then obtain g' **where** $g': \text{recfn } 1 g' \text{ total } g' \forall i. \text{eval } g' [i] = g i$
by *blast*
with $\langle ?P g \rangle$ **have** $?Q g'$ **by** *simp*
with *r-limr-def someI-ex*[$?Q$] **show**
 $\text{recfn } 1 r\text{-limr}$
 $\text{total } r\text{-limr}$
 $\varphi (\text{the } (\text{eval } r\text{-limr } [i])) \in \mathcal{R}$
 $\text{learn-lim } \psi U (\varphi i) \Longrightarrow \text{learn-lim } \psi U (\varphi (\text{the } (\text{eval } r\text{-limr } [i])))$
by *auto*
qed

For BC, too, *totalizer* works with every default hypothesis d .

lemma *lemma-R-for-BC*:
assumes $\text{learn-bc } \psi U (\varphi i)$
shows $\text{learn-bc } \psi U (\text{totalizer } d i)$
proof (*rule learn-bcI*)
show *env*: $\text{environment } \psi U (\text{totalizer } d i)$
using *assms learn-bcE(1) totalizer-in-R2* **by** *auto*
show $\exists n_0. \forall n \geq n_0. \psi (\text{the } (\text{totalizer } d i (f \triangleright n))) = f$ **if** $f \in U$ **for** f

proof –
have $f \in \mathcal{R}$
using *assms env that* **by** *auto*
obtain n_0 **where** $n_0: \forall n \geq n_0. \psi$ (*the* $((\varphi \ i) (f \triangleright n)) = f$)
using *assms learn-bcE* $\langle f \in U \rangle$ **by** *metis*
obtain m_0 **where** $m_0: \forall m \geq m_0. \text{the } (\text{delay } i (f \triangleright m)) > n_0$
using *delay-unbounded-monotone* $\langle f \in \mathcal{R} \rangle \langle f \in U \rangle$ *assms learn-bcE(1)*
by *blast*
then have $\forall m \geq m_0. \text{totalizer } d \ i (f \triangleright m) = \varphi \ i$ (*e-take* (*the* $(\text{delay } i (f \triangleright m))$)) $(f \triangleright m)$)
using *totalizer-def* **by** *auto*
then have $\forall m \geq m_0. \text{totalizer } d \ i (f \triangleright m) = \varphi \ i$ $(f \triangleright (\text{the } (\text{delay } i (f \triangleright m)) - 1))$
using *e-take-delay-init* $m_0 \langle f \in \mathcal{R} \rangle$ **by** *auto*
with $m_0 \ n_0$ **have** $\forall m \geq m_0. \psi$ (*the* $(\text{totalizer } d \ i (f \triangleright m)) = f$)
by *auto*
then show *?thesis* **by** *auto*
qed
qed

corollary *lemma-R-for-BC-simple:*

assumes *learn-bc* $\psi \ U \ s$
shows $\exists s' \in \mathcal{R}. \text{learn-bc } \psi \ U \ s'$
using *assms lemma-R-for-BC totalizer-in-R2 learn-bcE*
by (*metis R2-proj-R1 learn-bcE(1) phi-universal*)

For FIN the default hypothesis of *totalizer* must be zero, signalling “don’t know yet”.

lemma *lemma-R-for-FIN:*

assumes *learn-fin* $\psi \ U \ (\varphi \ i)$
shows *learn-fin* $\psi \ U \ (\text{totalizer } 0 \ i)$

proof (*rule learn-finI*)

show *env: environment* $\psi \ U \ (\text{totalizer } 0 \ i)$
using *assms learn-finE(1) totalizer-in-R2* **by** *auto*

show $\exists j \ n_0. \psi \ j = f \wedge$
 $(\forall n < n_0. \text{totalizer } 0 \ i (f \triangleright n) \downarrow = 0) \wedge$
 $(\forall n \geq n_0. \text{totalizer } 0 \ i (f \triangleright n) \downarrow = \text{Suc } j)$

if $f \in U$ **for** f

proof –

have $f \in \mathcal{R}$
using *assms env that* **by** *auto*
from *assms learn-finE*[*of* $\psi \ U \ \varphi \ i$] **obtain** j **where**
 $j: \psi \ j = f$ **and**
 $ex\text{-}n_0: \exists n_0. (\forall n < n_0. (\varphi \ i) (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. (\varphi \ i) (f \triangleright n) \downarrow = \text{Suc } j)$
using $\langle f \in U \rangle$ **by** *blast*
let $?Q = \lambda n_0. (\forall n < n_0. (\varphi \ i) (f \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. (\varphi \ i) (f \triangleright n) \downarrow = \text{Suc } j)$
define n_0 **where** $n_0 = \text{Least } ?Q$
with $ex\text{-}n_0$ **have** $n_0: ?Q \ n_0 \ \forall n < n_0. \neg ?Q \ n$
using *LeastI-ex*[*of* $?Q$] *not-less-Least*[*of* $- ?Q$] **by** *blast+*
define m_0 **where** $m_0 = (\text{LEAST } m_0. \forall m \geq m_0. \text{the } (\text{delay } i (f \triangleright m)) > n_0)$
(is $m_0 = \text{Least } ?P$ **)**
moreover have $\exists m_0. \forall m \geq m_0. \text{the } (\text{delay } i (f \triangleright m)) > n_0$
using *delay-unbounded-monotone* $\langle f \in \mathcal{R} \rangle \langle f \in U \rangle$ *assms learn-finE(1)*
by *simp*
ultimately have $m_0: ?P \ m_0 \ \forall m < m_0. \neg ?P \ m$
using *LeastI-ex*[*of* $?P$] *not-less-Least*[*of* $- ?P$] **by** *blast+*
then have $\forall m \geq m_0. \text{totalizer } 0 \ i (f \triangleright m) = \varphi \ i$ (*e-take* (*the* $(\text{delay } i (f \triangleright m))$)) $(f \triangleright m)$)
using *totalizer-def* **by** *auto*
then have $\forall m \geq m_0. \text{totalizer } 0 \ i (f \triangleright m) = \varphi \ i$ $(f \triangleright (\text{the } (\text{delay } i (f \triangleright m)) - 1))$

```

    using e-take-delay-init m0 ⟨f ∈ R⟩ by auto
with m0 n0 have ∀ m ≥ m0. totalizer 0 i (f ▷ m) ↓= Suc j
  by auto
moreover have totalizer 0 i (f ▷ m) ↓= 0 if m < m0 for m
proof (cases the (delay i (f ▷ m)) = 0)
  case True
  then show ?thesis by (simp add: totalizer-def)
next
  case False
  then have the (delay i (f ▷ m)) ≤ n0
    using m0 that ⟨f ∈ R⟩ delay-monotone by (meson leI order.strict-trans2)
  then show ?thesis
    using ⟨f ∈ R⟩ n0(1) totalizer-init by (simp add: Suc-le-lessD)
qed
ultimately show ?thesis using j by auto
qed
qed

```

2.8.2 Weaker Lemma R for CP and TOTAL

For TOTAL the default hypothesis used by *totalizer* depends on the hypothesis space, because it must refer to a total function in that space. Consequently the total strategy depends on the hypothesis space, which makes this form of Lemma R weaker than the ones in the previous section.

lemma *lemma-R-for-TOTAL*:

```

  fixes ψ :: partial2
  shows ∃ d. ∀ U. ∀ i. learn-total ψ U (φ i) ⟶ learn-total ψ U (totalizer d i)
proof (cases ∃ d. ψ d ∈ R)
  case True
  then obtain d where ψ d ∈ R by auto
  have learn-total ψ U (totalizer d i) if learn-total ψ U (φ i) for U i
  proof (rule learn-totalI)
    show env: environment ψ U (totalizer d i)
      using that learn-totalE(1) totalizer-in-R2 by auto
    show ∧f. f ∈ U ⟹ ∃ j. ψ j = f ∧ (∀ ∞ n. totalizer d i (f ▷ n) ↓= j)
      using that learn-total-def lemma-R-for-Lim[where ?d=d] learn-limE(2) by metis
    show ψ (the (totalizer d i (f ▷ n))) ∈ R if f ∈ U for f n
  proof (cases the (delay i (f ▷ n)) = 0)
    case True
    then show ?thesis using totalizer-def ⟨ψ d ∈ R⟩ by simp
  next
    case False
    have f ∈ R
      using that env by auto
    then show ?thesis
      using False that ⟨learn-total ψ U (φ i)⟩ totalizer-init learn-totalE(3)
      by simp
  qed
  qed
  then show ?thesis by auto
next
  case False
  then show ?thesis using learn-total-def lemma-R-for-Lim by auto
qed

```

corollary *lemma-R-for-TOTAL-simple*:
assumes *learn-total* ψ U s
shows $\exists s' \in \mathcal{R}. \text{learn-total } \psi \ U \ s'$
using *assms lemma-R-for-TOTAL totalizer-in-R2*
by (*metis R2-proj-R1 learn-totalE(1) phi-universal*)

For CP the default hypothesis used by *totalizer* depends on both the hypothesis space and the class. Therefore the total strategy depends on both the the hypothesis space and the class, which makes Lemma R for CP even weaker than the one for TOTAL.

lemma *lemma-R-for-CP*:
fixes $\psi :: \text{partial2}$ **and** $U :: \text{partial1 set}$
assumes *learn-cp* ψ U (φ i)
shows $\exists d. \text{learn-cp } \psi \ U \ (\text{totalizer } d \ i)$
proof (*cases* $U = \{\}$)
case *True*
then show *?thesis* **using** *assms learn-cp-def lemma-R-for-Lim* **by** *auto*
next
case *False*
then obtain f **where** $f \in U$ **by** *auto*
from $\langle f \in U \rangle$ **obtain** d **where** $\psi \ d = f$
using *learn-cpE(2)[OF assms]* **by** *auto*
with $\langle f \in U \rangle$ **have** $\psi \ d \in U$ **by** *simp*
have *learn-cp* ψ U (*totalizer* d i)
proof (*rule learn-cpI*)
show *env: environment* ψ U (*totalizer* d i)
using *assms learn-cpE(1) totalizer-in-R2* **by** *auto*
show $\bigwedge f. f \in U \implies \exists j. \psi \ j = f \wedge (\forall^\infty n. \text{totalizer } d \ i \ (f \triangleright n) \downarrow = j)$
using *assms learn-cp-def lemma-R-for-Lim[where ?d=d] learn-limE(2)* **by** *metis*
show ψ (*the* (*totalizer* d i ($f \triangleright n$))) $\in U$ **if** $f \in U$ **for** $f \ n$
proof (*cases the* (*delay* i ($f \triangleright n$)) = 0)
case *True*
then show *?thesis* **using** *totalizer-def* $\langle \psi \ d \in U \rangle$ **by** *simp*
next
case *False*
then show *?thesis*
using *that env assms totalizer-init learn-cpE(3)* **by** *auto*
qed
qed
then show *?thesis* **by** *auto*
qed

2.8.3 No Lemma R for CONS

This section demonstrates that the class V_{01} of all total recursive functions f where $f(0)$ or $f(1)$ is a Gödel number of f can be consistently learned in the limit, but not by a total strategy. This implies that Lemma R does not hold for CONS.

definition $V_{01} :: \text{partial1 set } (\langle V_{01} \rangle)$ **where**
 $V_{01} = \{f. f \in \mathcal{R} \wedge (\varphi \ (\text{the } (f \ 0)) = f \vee \varphi \ (\text{the } (f \ 1)) = f)\}$

No total CONS strategy for V_{01}

In order to show that no total strategy can learn V_{01} we construct, for each total strategy S , one or two functions in V_{01} such that S fails for at least one of them. At the core

of this construction is a process that given a total recursive strategy S and numbers $z, i, j \in \mathbb{N}$ builds a function f as follows: Set $f(0) = i$ and $f(1) = j$. For $x \geq 1$:

- (a) Check whether S changes its hypothesis when f^x is extended by 0, that is, if $S(f^x) \neq S(f^x0)$. If so, set $f(x+1) = 0$.
- (b) Otherwise check if S changes its hypothesis when f^x is extended by 1, that is, if $S(f^x) \neq S(f^x1)$. If so, set $f(x+1) = 1$.
- (c) If neither happens, set $f(x+1) = z$.

In other words, as long as we can force S to change its hypothesis by extending the function by 0 or 1, we do just that. Now there are two cases:

- Case 1. For all $x \geq 1$ either (a) or (b) occurs; then S changes its hypothesis on f all the time and thus does not learn f in the limit (not to mention consistently). The value of z makes no difference in this case.
- Case 2. For some minimal x , (c) occurs, that is, there is an f^x such that $h := S(f^x) = S(f^x0) = S(f^x1)$. But the hypothesis h cannot be consistent with both prefixes f^x0 and f^x1 . Running the process once with $z = 0$ and once with $z = 1$ yields two functions starting with f^x0 and f^x1 , respectively, such that S outputs the same hypothesis, h , on both prefixes and thus cannot be consistent for both functions.

This process is computable because S is total. The construction does not work if we only assume S to be a CONS strategy for V_{01} , because we need to be able to apply S to prefixes not in V_{01} .

The parameters i and j provide flexibility to find functions built by the above process that are actually in V_{01} . To this end we will use Smullyan's double fixed-point theorem.

context

fixes $s :: \text{partial1}$

assumes $s\text{-in-}R1$ [*simp*, *intro*]: $s \in \mathcal{R}$

begin

The function *prefixes* constructs prefixes according to the aforementioned process.

fun *prefixes* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list}$ **where**

prefixes $z\ i\ j\ 0 = [i]$

| *prefixes* $z\ i\ j\ (\text{Suc } x) = \text{prefixes } z\ i\ j\ x\ @$

[*if* $x = 0$ *then* j

else if $s\ (\text{list-encode } (\text{prefixes } z\ i\ j\ x\ @\ [0])) \neq s\ (\text{list-encode } (\text{prefixes } z\ i\ j\ x))$

then 0

else if $s\ (\text{list-encode } (\text{prefixes } z\ i\ j\ x\ @\ [1])) \neq s\ (\text{list-encode } (\text{prefixes } z\ i\ j\ x))$

then 1

else $z]$

lemma *prefixes-length*: $\text{length } (\text{prefixes } z\ i\ j\ x) = \text{Suc } x$

by (*induction* x) *simp-all*

The functions *adverse* $z\ i\ j$ are the functions constructed by *prefixes*.

definition *adverse* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat option}$ **where**

adverse $z\ i\ j\ x \equiv \text{Some } (\text{last } (\text{prefixes } z\ i\ j\ x))$

lemma *init-adverse-eq-prefixes*: $(\text{adverse } z\ i\ j) \triangleright n = \text{list-encode } (\text{prefixes } z\ i\ j\ n)$

```

proof –
  have prefix (adverse z i j) n = prefixes z i j n
  proof (induction n)
    case 0
    then show ?case using adverse-def prefixes-length prefixI' by fastforce
  next
    case (Suc n)
    then show ?case using adverse-def by (simp add: prefix-Suc)
  qed
  then show ?thesis by (simp add: init-def)
qed

```

```

lemma adverse-at-01:
  adverse z i j 0  $\downarrow$ = i
  adverse z i j 1  $\downarrow$ = j
  by (auto simp add: adverse-def)

```

Had we introduced ternary partial recursive functions, the *adverse* *z* functions would be among them.

```

lemma adverse-in-R3:  $\exists r. \text{recfn } 3 r \wedge \text{total } r \wedge (\lambda i j x. \text{eval } r [i, j, x]) = \text{adverse } z$ 

```

```

proof –
  obtain rs where rs: recfn 1 rs total rs  $(\lambda x. \text{eval } rs [x]) = s$ 
  using R1E by auto
  have s-total:  $\bigwedge x. s x \downarrow$  by simp

```

```

define f where f = Cn 2 r-singleton-encode [Id 2 0]
then have recfn 2 f by simp
have f:  $\bigwedge i j. \text{eval } f [i, j] \downarrow = \text{list-encode } [i]$ 
  unfolding f-def by simp

```

```

define ch1 where ch1 = Cn 4 r-ifeq
  [Cn 4 rs [Cn 4 r-snoc [Id 4 1, r-constn 3 1]],
  Cn 4 rs [Id 4 1],
  r-dummy 3 (r-const z),
  r-constn 3 1]
then have ch1: recfn 4 ch1 total ch1
  using Cn-total prim-recfn-total rs by auto

```

```

define ch0 where ch0 = Cn 4 r-ifeq
  [Cn 4 rs [Cn 4 r-snoc [Id 4 1, r-constn 3 0]],
  Cn 4 rs [Id 4 1],
  ch1,
  r-constn 3 0]
then have ch0-total: total ch0 recfn 4 ch0
  using Cn-total prim-recfn-total rs ch1 by auto

```

```

have eval ch1 [l, v, i, j]  $\downarrow = (\text{if } s (e\text{-snoc } v 1) = s v \text{ then } z \text{ else } 1)$  for l v i j

```

```

proof –
  have eval ch1 [l, v, i, j] = eval r-ifeq [the (s (e-snoc v 1)), the (s v), z, 1]
  unfolding ch1-def using rs by auto
  then show ?thesis by (simp add: s-total option.expand)

```

```

qed

```

```

moreover have eval ch0 [l, v, i, j]  $\downarrow =$ 
  (if s (e-snoc v 0) = s v then the (eval ch1 [l, v, i, j]) else 0) for l v i j

```

```

proof –
  have eval ch0 [l, v, i, j] =

```

```

    eval r-ifeq [the (s (e-snoc v 0)), the (s v), the (eval ch1 [l, v, i, j]), 0]
  unfolding ch0-def using rs ch1 by auto
  then show ?thesis by (simp add: s-total option.expand)
qed
ultimately have ch0:  $\bigwedge l v i j. \text{eval } ch0 [l, v, i, j] \downarrow =$ 
  (if s (e-snoc v 0)  $\neq$  s v then 0
   else if s (e-snoc v 1)  $\neq$  s v then 1 else z)
  by simp

define app where app = Cn 4 r-ifz [Id 4 0, Id 4 3, ch0]
then have recfn 4 app total app
  using ch0-total totalI4 by auto
have eval app [l, v, i, j]  $\downarrow =$  (if l = 0 then j else the (eval ch0 [l, v, i, j])) for l v i j
  unfolding app-def using ch0-total by simp
with ch0 have app:  $\bigwedge l v i j. \text{eval } app [l, v, i, j] \downarrow =$ 
  (if l = 0 then j
   else if s (e-snoc v 0)  $\neq$  s v then 0
   else if s (e-snoc v 1)  $\neq$  s v then 1 else z)
  by simp

define g where g = Cn 4 r-snoc [Id 4 1, app]
with app have g:  $\bigwedge l v i j. \text{eval } g [l, v, i, j] \downarrow = \text{e-snoc } v$ 
  (if l = 0 then j
   else if s (e-snoc v 0)  $\neq$  s v then 0
   else if s (e-snoc v 1)  $\neq$  s v then 1 else z)
  using <recfn 4 app> by auto
from g-def have recfn 4 g total g
  using <recfn 4 app> <total app> Cn-total Mn-free-imp-total by auto

define b where b = Pr 2 f g
then have recfn 3 b
  using <recfn 2 f> <recfn 4 g> by simp
have b: eval b [x, i, j]  $\downarrow = \text{list-encode (prefixes } z \text{ } i \text{ } j \text{ } x)$  for x i j
proof (induction x)
  case 0
  then show ?case
    unfolding b-def using f <recfn 2 f> <recfn 4 g> by simp
next
  case (Suc x)
  then have eval b [Suc x, i, j] = eval g [x, the (eval b [x, i, j]), i, j]
    using b-def <recfn 3 b> by simp
  also have ...  $\downarrow =$ 
    (let v = list-encode (prefixes z i j x)
     in e-snoc v
     (if x = 0 then j
      else if s (e-snoc v 0)  $\neq$  s v then 0
      else if s (e-snoc v 1)  $\neq$  s v then 1 else z))
    using g Suc by simp
  also have ...  $\downarrow =$ 
    (let v = list-encode (prefixes z i j x)
     in e-snoc v
     (if x = 0 then j
      else if s (list-encode (prefixes z i j x @ [0]))  $\neq$  s v then 0
      else if s (list-encode (prefixes z i j x @ [1]))  $\neq$  s v then 1 else z))
    using list-decode-encode by presburger
  finally show ?case by simp

```

qed

define b' **where** $b' = Cn\ 3\ b\ [Id\ 3\ 2,\ Id\ 3\ 0,\ Id\ 3\ 1]$
then have $recfn\ 3\ b'$
 using $\langle recfn\ 3\ b \rangle$ **by** *simp*
with b **have** $b': \bigwedge i\ j\ x. eval\ b'\ [i,\ j,\ x] \Downarrow = list-encode\ (prefixes\ z\ i\ j\ x)$
 using $b'-def$ **by** *simp*

define r **where** $r = Cn\ 3\ r-last\ [b']$
then have $recfn\ 3\ r$
 using $\langle recfn\ 3\ b' \rangle$ **by** *simp*
with b' **have** $\bigwedge i\ j\ x. eval\ r\ [i,\ j,\ x] \Downarrow = last\ (prefixes\ z\ i\ j\ x)$
 using $r-def\ prefixes-length$ **by** *auto*
moreover from this have $total\ r$
 using $totalI3\ \langle recfn\ 3\ r \rangle$ **by** *simp*
ultimately have $(\lambda i\ j\ x. eval\ r\ [i,\ j,\ x]) = adverse\ z$
 unfolding $adverse-def$ **by** *simp*
with $\langle recfn\ 3\ r \rangle\ \langle total\ r \rangle$ **show** $?thesis$ **by** *auto*
qed

lemma *adverse-in-R1*: $adverse\ z\ i\ j \in \mathcal{R}$

proof –

from *adverse-in-R3* **obtain** r **where**
 $r: recfn\ 3\ r\ total\ r\ (\lambda i\ j\ x. eval\ r\ [i,\ j,\ x]) = adverse\ z$
 by *blast*
define rij **where** $rij = Cn\ 1\ r\ [r-const\ i,\ r-const\ j,\ Id\ 1\ 0]$
then have $recfn\ 1\ rij\ total\ rij$
 using $r(1,2)\ Cn-total\ Mn-free-imp-total$ **by** *auto*
from $rij-def$ **have** $\bigwedge x. eval\ rij\ [x] = eval\ r\ [i,\ j,\ x]$
 using $r(1)$ **by** *auto*
with $r(3)$ **have** $\bigwedge x. eval\ rij\ [x] = adverse\ z\ i\ j\ x$
 by *metis*
with $\langle recfn\ 1\ rij \rangle\ \langle total\ rij \rangle$ **show** $?thesis$ **by** *auto*
qed

Next we show that for every z there are i, j such that $adverse\ z\ i\ j \in V_{01}$. The first step is to show that for every z , Gödel numbers for $adverse\ z\ i\ j$ can be computed uniformly from i and j .

lemma *phi-translate-adverse*: $\exists f \in \mathcal{R}^2. \forall i\ j. \varphi\ (the\ (f\ i\ j)) = adverse\ z\ i\ j$

proof –

obtain r **where** $r: recfn\ 3\ r\ total\ r\ (\lambda i\ j\ x. eval\ r\ [i,\ j,\ x]) = adverse\ z$
 using *adverse-in-R3* **by** *blast*
let $?p = encode\ r$
define rf **where** $rf = Cn\ 2\ (r-smn\ 1\ 2)\ [r-dummy\ 1\ (r-const\ ?p),\ Id\ 2\ 0,\ Id\ 2\ 1]$
then have $recfn\ 2\ rf$ **and** $total\ rf$
 using $Mn-free-imp-total$ **by** *simp-all*
define f **where** $f \equiv \lambda i\ j. eval\ rf\ [i,\ j]$
with $\langle recfn\ 2\ rf \rangle\ \langle total\ rf \rangle$ **have** $f \in \mathcal{R}^2$ **by** *auto*
have $rf: eval\ rf\ [i,\ j] = eval\ (r-smn\ 1\ 2)\ [?p,\ i,\ j]$ **for** $i\ j$
 unfolding $rf-def$ **by** *simp*
{
 fix $i\ j\ x$
 have $\varphi\ (the\ (f\ i\ j))\ x = eval\ r-phi\ [the\ (f\ i\ j),\ x]$
 using $phi-def$ **by** *simp*
 also have $\dots = eval\ r-phi\ [the\ (eval\ rf\ [i,\ j]),\ x]$
 using $f-def$ **by** *simp*

```

also have ... = eval (r-universal 1) [the (eval (r-smn 1 2) [?p, i, j]), x]
  using rf r-phi-def by simp
also have ... = eval (r-universal (2 + 1)) (?p # [i, j] @ [x])
  using smn-lemma[of 1 [i, j] 2 [x]] by simp
also have ... = eval (r-universal 3) [?p, i, j, x]
  by simp
also have ... = eval r [i, j, x]
  using r-universal r by force
also have ... = adverse z i j x
  using r(3) by metis
finally have  $\varphi$  (the (f i j)) x = adverse z i j x .
}
with  $\langle f \in \mathcal{R}^2 \rangle$  show ?thesis by blast
qed

```

The second, and final, step is to apply Smullyan's double fixed-point theorem to show the existence of *adverse* functions in V_{01} .

lemma *adverse-in-V01*: $\exists m n. \text{adverse } 0 m n \in V_{01} \wedge \text{adverse } 1 m n \in V_{01}$

proof –

```

obtain f0 where f0: f0 ∈  $\mathcal{R}^2 \forall i j. \varphi$  (the (f0 i j)) = adverse 0 i j
  using phi-translate-adverse[of 0] by auto
obtain f1 where f1: f1 ∈  $\mathcal{R}^2 \forall i j. \varphi$  (the (f1 i j)) = adverse 1 i j
  using phi-translate-adverse[of 1] by auto
obtain m n where  $\varphi m = \varphi$  (the (f0 m n)) and  $\varphi n = \varphi$  (the (f1 m n))
  using smullyan-double-fixed-point[OF f0(1) f1(1)] by blast
with f0(2) f1(2) have  $\varphi m = \text{adverse } 0 m n$  and  $\varphi n = \text{adverse } 1 m n$ 
  by simp-all
moreover have the (adverse 0 m n 0) = m and the (adverse 1 m n 1) = n
  using adverse-at-01 by simp-all
ultimately have
   $\varphi$  (the (adverse 0 m n 0)) = adverse 0 m n
   $\varphi$  (the (adverse 1 m n 1)) = adverse 1 m n
  by simp-all
moreover have adverse 0 m n ∈  $\mathcal{R}$  and adverse 1 m n ∈  $\mathcal{R}$ 
  using adverse-in-R1 by simp-all
ultimately show ?thesis using V01-def by auto

```

qed

Before we prove the main result of this section we need some lemmas regarding the shape of the *adverse* functions and hypothesis changes of the strategy.

lemma *adverse-Suc*:

```

assumes x > 0
shows adverse z i j (Suc x)  $\downarrow$ =
  (if s (e-snoc ((adverse z i j)  $\triangleright$  x) 0)  $\neq$  s ((adverse z i j)  $\triangleright$  x)
   then 0
   else if s (e-snoc ((adverse z i j)  $\triangleright$  x) 1)  $\neq$  s ((adverse z i j)  $\triangleright$  x)
   then 1 else z)

```

proof –

```

have adverse z i j (Suc x)  $\downarrow$ =
  (if s (list-encode (prefixes z i j x @ [0]))  $\neq$  s (list-encode (prefixes z i j x))
   then 0
   else if s (list-encode (prefixes z i j x @ [1]))  $\neq$  s (list-encode (prefixes z i j x))
   then 1 else z)
  using assms adverse-def by simp
then show ?thesis by (simp add: init-adverse-eq-prefixes)

```

qed

The process in the proof sketch (page 168) consists of steps (a), (b), and (c). The next abbreviation is true iff. step (a) or (b) applies.

abbreviation $\text{hyp-change } z \ i \ j \ x \equiv$
 $s \ (e\text{-snoc } ((\text{adverse } z \ i \ j) \triangleright x) \ 0) \neq s \ ((\text{adverse } z \ i \ j) \triangleright x) \vee$
 $s \ (e\text{-snoc } ((\text{adverse } z \ i \ j) \triangleright x) \ 1) \neq s \ ((\text{adverse } z \ i \ j) \triangleright x)$

If step (c) applies, the process appends z .

lemma *adverse-Suc-not-hyp-change*:
assumes $x > 0$ **and** $\neg \text{hyp-change } z \ i \ j \ x$
shows $\text{adverse } z \ i \ j \ (\text{Suc } x) \downarrow = z$
using *assms* *adverse-Suc* **by** *simp*

While (a) or (b) applies, the process appends a value that forces S to change its hypothesis.

lemma *while-hyp-change*:
assumes $\forall x \leq n. x > 0 \longrightarrow \text{hyp-change } z \ i \ j \ x$
shows $\forall x \leq \text{Suc } n. \text{adverse } z \ i \ j \ x = \text{adverse } z' \ i \ j \ x$
using *assms*
proof (*induction n*)
case 0
then show *?case* **by** (*simp add: adverse-def le-Suc-eq*)
next
case (*Suc n*)
then have $\forall x \leq n. x > 0 \longrightarrow \text{hyp-change } z \ i \ j \ x$ **by** *simp*
with *Suc* **have** $\forall x \leq \text{Suc } n. x > 0 \longrightarrow \text{adverse } z \ i \ j \ x = \text{adverse } z' \ i \ j \ x$
by *simp*
moreover have $\text{adverse } z \ i \ j \ 0 = \text{adverse } z' \ i \ j \ 0$
using *adverse-at-01* **by** *simp*
ultimately have *zz'*: $\forall x \leq \text{Suc } n. \text{adverse } z \ i \ j \ x = \text{adverse } z' \ i \ j \ x$
by *auto*
moreover have $\text{adverse } z \ i \ j \in \mathcal{R} \ \text{adverse } z' \ i \ j \in \mathcal{R}$
using *adverse-in-R1* **by** *simp-all*
ultimately have *init-zz'*: $(\text{adverse } z \ i \ j) \triangleright (\text{Suc } n) = (\text{adverse } z' \ i \ j) \triangleright (\text{Suc } n)$
using *init-eqI* **by** *blast*

have $\text{adverse } z \ i \ j \ (\text{Suc } (\text{Suc } n)) = \text{adverse } z' \ i \ j \ (\text{Suc } (\text{Suc } n))$
proof (*cases s (e-snoc ((adverse z i j) ▷ (Suc n)) 0) ≠ s ((adverse z i j) ▷ (Suc n))*)
case *True*
then have $s \ (e\text{-snoc } ((\text{adverse } z' \ i \ j) \triangleright (\text{Suc } n)) \ 0) \neq s \ ((\text{adverse } z' \ i \ j) \triangleright (\text{Suc } n))$
using *init-zz'* **by** *simp*
then have $\text{adverse } z' \ i \ j \ (\text{Suc } (\text{Suc } n)) \downarrow = 0$
by (*simp add: adverse-Suc*)
moreover have $\text{adverse } z \ i \ j \ (\text{Suc } (\text{Suc } n)) \downarrow = 0$
using *True* **by** (*simp add: adverse-Suc*)
ultimately show *?thesis* **by** *simp*
next
case *False*
then have $s \ (e\text{-snoc } ((\text{adverse } z' \ i \ j) \triangleright (\text{Suc } n)) \ 0) = s \ ((\text{adverse } z' \ i \ j) \triangleright (\text{Suc } n))$
using *init-zz'* **by** *simp*
then have $\text{adverse } z' \ i \ j \ (\text{Suc } (\text{Suc } n)) \downarrow = 1$
using *init-zz' Suc.premis adverse-Suc* **by** (*smt le-refl zero-less-Suc*)
moreover have $\text{adverse } z \ i \ j \ (\text{Suc } (\text{Suc } n)) \downarrow = 1$
using *False Suc.premis adverse-Suc* **by** *auto*

ultimately show *?thesis* by *simp*
 qed
 with *zz'* show *?case* using *le-SucE* by *blast*
 qed

The next result corresponds to Case 1 from the proof sketch.

lemma *always-hyp-change-no-lim*:
 assumes $\forall x > 0. \text{hyp-change } z \ i \ j \ x$
 shows $\neg \text{learn-lim } \varphi \ \{ \text{adverse } z \ i \ j \} \ s$
proof (*rule infinite-hyp-changes-not-Lim*[of *adverse z i j*])
 show *adverse z i j* $\in \{ \text{adverse } z \ i \ j \}$ by *simp*
 show $\forall n. \exists m_1 > n. \exists m_2 > n. s \ (\text{adverse } z \ i \ j \triangleright m_1) \neq s \ (\text{adverse } z \ i \ j \triangleright m_2)$
proof
 fix *n*
 from *assms* obtain *m1* where *m1*: $m_1 > n$ *hyp-change z i j m1*
 by *auto*
 have $s \ (\text{adverse } z \ i \ j \triangleright m_1) \neq s \ (\text{adverse } z \ i \ j \triangleright (\text{Suc } m_1))$
proof (*cases s (e-snoc ((adverse z i j) \triangleright m1) 0) \neq s ((adverse z i j) \triangleright m1)*)
 case *True*
 then have *adverse z i j (Suc m1)* $\downarrow = 0$
 using *m1 adverse-Suc* by *simp*
 then have $(\text{adverse } z \ i \ j) \triangleright (\text{Suc } m_1) = e\text{-snoc } ((\text{adverse } z \ i \ j) \triangleright m_1) \ 0$
 by (*simp add: init-Suc-snoc*)
 with *True* show *?thesis* by *simp*
 next
 case *False*
 then have *adverse z i j (Suc m1)* $\downarrow = 1$
 using *m1 adverse-Suc* by *simp*
 then have $(\text{adverse } z \ i \ j) \triangleright (\text{Suc } m_1) = e\text{-snoc } ((\text{adverse } z \ i \ j) \triangleright m_1) \ 1$
 by (*simp add: init-Suc-snoc*)
 with *False m1* (2) show *?thesis* by *simp*
 qed
 then show $\exists m_1 > n. \exists m_2 > n. s \ (\text{adverse } z \ i \ j \triangleright m_1) \neq s \ (\text{adverse } z \ i \ j \triangleright m_2)$
 using *less-SucI m1* (1) by *blast*
 qed
 qed

The next result corresponds to Case 2 from the proof sketch.

lemma *no-hyp-change-no-cons*:
 assumes $x > 0$ and $\neg \text{hyp-change } z \ i \ j \ x$
 shows $\neg \text{learn-cons } \varphi \ \{ \text{adverse } 0 \ i \ j, \text{adverse } 1 \ i \ j \} \ s$
proof –
 let *?P* = $\lambda x. x > 0 \wedge \neg \text{hyp-change } z \ i \ j \ x$
 define *xmin* where *xmin* = *Least ?P*
 with *assms* have *xmin*:
?P xmin
 $\bigwedge x. x < \text{xmin} \implies \neg ?P \ x$
 using *LeastI*[of *?P*] *not-less-Least*[of *- ?P*] by *simp-all*
 then have *xmin* > 0 by *simp*

 have $\forall x \leq \text{xmin} - 1. x > 0 \implies \text{hyp-change } z \ i \ j \ x$
 using *xmin* by (*metis One-nat-def Suc-pred le-imp-less-Suc*)
 then have
 $\forall x \leq \text{xmin}. \text{adverse } z \ i \ j \ x = \text{adverse } 0 \ i \ j \ x$
 $\forall x \leq \text{xmin}. \text{adverse } z \ i \ j \ x = \text{adverse } 1 \ i \ j \ x$
 using *while-hyp-change*[of *xmin - 1 z i j 0*]

```

using while-hyp-change[of xmin - 1 z i j 1]
by simp-all
then have
  init-z0: (adverse z i j) ▷ xmin = (adverse 0 i j) ▷ xmin and
  init-z1: (adverse z i j) ▷ xmin = (adverse 1 i j) ▷ xmin
using adverse-in-R1 init-eqI by blast+
then have
  a0: adverse 0 i j (Suc xmin) ↓= 0 and
  a1: adverse 1 i j (Suc xmin) ↓= 1
using adverse-Suc-not-hyp-change xmin(1) init-z1
bymetis+
then have
  i0: (adverse 0 i j) ▷ (Suc xmin) = e-snoc ((adverse z i j) ▷ xmin) 0 and
  i1: (adverse 1 i j) ▷ (Suc xmin) = e-snoc ((adverse z i j) ▷ xmin) 1
using init-z0 init-z1 by (simp-all add: init-Suc-snoc)
moreover have
  s (e-snoc ((adverse z i j) ▷ xmin) 0) = s ((adverse z i j) ▷ xmin)
  s (e-snoc ((adverse z i j) ▷ xmin) 1) = s ((adverse z i j) ▷ xmin)
using xmin by simp-all
ultimately have
  s ((adverse 0 i j) ▷ (Suc xmin)) = s ((adverse z i j) ▷ xmin)
  s ((adverse 1 i j) ▷ (Suc xmin)) = s ((adverse z i j) ▷ xmin)
by simp-all
then have
  s ((adverse 0 i j) ▷ (Suc xmin)) = s ((adverse 1 i j) ▷ (Suc xmin))
by simp
moreover have (adverse 0 i j) ▷ (Suc xmin) ≠ (adverse 1 i j) ▷ (Suc xmin)
using a0 a1 i0 i1 by (metis append1-eq-conv list-decode-encode zero-neq-one)
ultimately show ¬ learn-cons φ {adverse 0 i j, adverse 1 i j} s
using same-hyp-different-init-not-cons by blast
qed

```

Combining the previous two lemmas shows that V_{01} cannot be learned consistently in the limit by the total strategy S .

lemma *V01-not-in-R-cons*: ¬ learn-cons φ V_{01} s

proof –

obtain m n **where**

mn0: adverse 0 m n ∈ V_{01} **and**

mn1: adverse 1 m n ∈ V_{01}

using adverse-in-V01 **by** auto

show ¬ learn-cons φ V_{01} s

proof (cases ∀x>0. hyp-change 0 m n x)

case True

then have ¬ learn-lim φ {adverse 0 m n} s

using always-hyp-change-no-lim **by** simp

with mn0 **show** ?thesis

using learn-cons-def learn-lim-closed-subseteq **by** auto

next

case False

then obtain x **where** x: x > 0 ¬ hyp-change 0 m n x **by** auto

then have ¬ learn-cons φ {adverse 0 m n, adverse 1 m n} s

using no-hyp-change-no-cons[OF x] **by** simp

with mn0 mn1 **show** ?thesis **using** learn-cons-closed-subseteq **by** auto

qed

qed

end

V_{01} is in CONS

At first glance, consistently learning V_{01} looks fairly easy. After all every $f \in V_{01}$ provides a Gödel number of itself either at argument 0 or 1. A strategy only has to figure out which one is right. However, the strategy S we are going to devise does not always converge to $f(0)$ or $f(1)$. Instead it uses a technique called “amalgamation”. The amalgamation of two Gödel numbers i and j is a function whose value at x is determined by simulating $\varphi_i(x)$ and $\varphi_j(x)$ in parallel and outputting the value of the first one to halt. If neither halts the value is undefined. There is a function $a \in \mathcal{R}^2$ such that $\varphi_{a(i,j)}$ is the amalgamation of i and j .

If $f \in V_{01}$ then $\varphi_{a(f(0),f(1))}$ is total because by definition of V_{01} we have $\varphi_{f(0)} = f$ or $\varphi_{f(1)} = f$ and f is total.

Given a prefix f^n of an $f \in V_{01}$ the strategy S first computes $\varphi_{a(f(0),f(1))}(x)$ for $x = 0, \dots, n$. For the resulting prefix $\varphi_{a(f(0),f(1))}^n$ there are two cases:

- Case 1. It differs from f^n , say at minimum index x . Then for either $z = 0$ or $z = 1$ we have $\varphi_{f(z)}(x) \neq f(x)$ by definition of amalgamation. This implies $\varphi_{f(z)} \neq f$, and thus $\varphi_{f(1-z)} = f$ by definition of V_{01} . We set $S(f^n) = f(1-z)$. This hypothesis is correct and hence consistent.
- Case 2. It equals f^n . Then we set $S(f^n) = a(f(0), f(1))$. This hypothesis is consistent by definition of this case.

In both cases the hypothesis is consistent. If Case 1 holds for some n , the same x and z will be found also for all larger values of n . Therefore S converges to the correct hypothesis $f(1-z)$. If Case 2 holds for all n , then S always outputs the same hypothesis $a(f(0), f(1))$ and thus also converges.

The above discussion tacitly assumes $n \geq 1$, such that both $f(0)$ and $f(1)$ are available to S . For $n = 0$ the strategy outputs an arbitrary consistent hypothesis.

Amalgamation uses the concurrent simulation of functions.

definition *parallel* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$ option **where**
parallel i j $x \equiv eval$ *r-parallel* $[i, j, x]$

lemma *r-parallel'*: $eval$ *r-parallel* $[i, j, x] = parallel$ i j x
using *parallel-def* **by** *simp*

lemma *r-parallel''*:

shows $eval$ *r-phi* $[i, x] \uparrow \wedge eval$ *r-phi* $[j, x] \uparrow \implies eval$ *r-parallel* $[i, j, x] \uparrow$
and $eval$ *r-phi* $[i, x] \downarrow \wedge eval$ *r-phi* $[j, x] \uparrow \implies$
 $eval$ *r-parallel* $[i, j, x] \downarrow = prod$ -*encode* $(0, the (eval$ *r-phi* $[i, x]))$
and $eval$ *r-phi* $[j, x] \downarrow \wedge eval$ *r-phi* $[i, x] \uparrow \implies$
 $eval$ *r-parallel* $[i, j, x] \downarrow = prod$ -*encode* $(1, the (eval$ *r-phi* $[j, x]))$
and $eval$ *r-phi* $[i, x] \downarrow \wedge eval$ *r-phi* $[j, x] \downarrow \implies$
 $eval$ *r-parallel* $[i, j, x] \downarrow = prod$ -*encode* $(0, the (eval$ *r-phi* $[i, x])) \vee$
 $eval$ *r-parallel* $[i, j, x] \downarrow = prod$ -*encode* $(1, the (eval$ *r-phi* $[j, x]))$

proof –

let $?f = Cn$ 1 *r-phi* $[r$ -*const* i, Id 1 $0]$
let $?g = Cn$ 1 *r-phi* $[r$ -*const* j, Id 1 $0]$
have $*$: $\bigwedge x. eval$ *r-phi* $[i, x] = eval$ $?f$ $[x] \wedge x. eval$ *r-phi* $[j, x] = eval$ $?g$ $[x]$

by *simp-all*
show $\text{eval } r\text{-phi } [i, x] \uparrow \wedge \text{eval } r\text{-phi } [j, x] \uparrow \implies \text{eval } r\text{-parallel } [i, j, x] \uparrow$
and $\text{eval } r\text{-phi } [i, x] \downarrow \wedge \text{eval } r\text{-phi } [j, x] \uparrow \implies$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } r\text{-phi } [i, x]))$
and $\text{eval } r\text{-phi } [j, x] \downarrow \wedge \text{eval } r\text{-phi } [i, x] \uparrow \implies$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (1, \text{the } (\text{eval } r\text{-phi } [j, x]))$
and $\text{eval } r\text{-phi } [i, x] \downarrow \wedge \text{eval } r\text{-phi } [j, x] \downarrow \implies$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (0, \text{the } (\text{eval } r\text{-phi } [i, x])) \vee$
 $\text{eval } r\text{-parallel } [i, j, x] \downarrow = \text{prod-encode } (1, \text{the } (\text{eval } r\text{-phi } [j, x]))$
using $r\text{-parallel}[OF *]$ **by** *simp-all*
qed

lemma *parallel*:

$\varphi i x \uparrow \wedge \varphi j x \uparrow \implies \text{parallel } i j x \uparrow$
 $\varphi i x \downarrow \wedge \varphi j x \uparrow \implies \text{parallel } i j x \downarrow = \text{prod-encode } (0, \text{the } (\varphi i x))$
 $\varphi j x \downarrow \wedge \varphi i x \uparrow \implies \text{parallel } i j x \downarrow = \text{prod-encode } (1, \text{the } (\varphi j x))$
 $\varphi i x \downarrow \wedge \varphi j x \downarrow \implies$
 $\text{parallel } i j x \downarrow = \text{prod-encode } (0, \text{the } (\varphi i x)) \vee$
 $\text{parallel } i j x \downarrow = \text{prod-encode } (1, \text{the } (\varphi j x))$
using $\text{phi-def } r\text{-parallel''}$ $r\text{-parallel}$ parallel-def **by** *simp-all*

lemma *parallel-converge-pdec1-0-or-1*:

assumes $\text{parallel } i j x \downarrow$
shows $\text{pdec1 } (\text{the } (\text{parallel } i j x)) = 0 \vee \text{pdec1 } (\text{the } (\text{parallel } i j x)) = 1$
using $\text{assms } \text{parallel}[of i x j]$ $\text{parallel}(3)[of j x i]$
by (*metis fst-eqD option.sel prod-encode-inverse*)

lemma *parallel-converge-either*: $(\varphi i x \downarrow \vee \varphi j x \downarrow) = (\text{parallel } i j x \downarrow)$
using parallel **by** (*metis option.simps(3)*)

lemma *parallel-0*:

assumes $\text{parallel } i j x \downarrow = \text{prod-encode } (0, v)$
shows $\varphi i x \downarrow = v$
using parallel assms
by (*smt option.collapse option.sel option.simps(3) prod.inject prod-encode-eq zero-neq-one*)

lemma *parallel-1*:

assumes $\text{parallel } i j x \downarrow = \text{prod-encode } (1, v)$
shows $\varphi j x \downarrow = v$
using parallel assms
by (*smt option.collapse option.sel option.simps(3) prod.inject prod-encode-eq zero-neq-one*)

lemma *parallel-converge-V01*:

assumes $f \in V_{01}$
shows $\text{parallel } (\text{the } (f 0)) (\text{the } (f 1)) x \downarrow$
proof –
have $f \in \mathcal{R} \wedge (\varphi (\text{the } (f 0)) = f \vee \varphi (\text{the } (f 1)) = f)$
using $\text{assms } V01\text{-def}$ **by** *auto*
then have $\varphi (\text{the } (f 0)) \in \mathcal{R} \vee \varphi (\text{the } (f 1)) \in \mathcal{R}$
by *auto*
then have $\varphi (\text{the } (f 0)) x \downarrow \vee \varphi (\text{the } (f 1)) x \downarrow$
using $R1\text{-imp-total1}$ **by** *auto*
then show *thesis* **using** *parallel-converge-either* **by** *simp*
qed

The amalgamation of two Gödel numbers can then be described in terms of *parallel*.

definition *amalgamation* :: nat ⇒ nat ⇒ partial1 **where**
amalgamation *i j* ≡
 if parallel *i j x* ↑ then None else Some (pdec2 (the (parallel *i j x*)))

lemma *amalgamation-diverg*: *amalgamation i j x* ↑ ⇔ φ *i x* ↑ ∧ φ *j x* ↑
using *amalgamation-def parallel* **by** (metis option.simps(3))

lemma *amalgamation-total*:
assumes total1 (φ *i*) ∨ total1 (φ *j*)
shows total1 (*amalgamation i j*)
using *assms amalgamation-diverg*[of *i j*] *total-def* **by** auto

lemma *amalgamation-V01-total*:
assumes *f* ∈ *V*₀₁
shows total1 (*amalgamation* (the (*f* 0)) (the (*f* 1)))
using *assms V01-def amalgamation-total R1-imp-total1 total1-def*
by (metis (mono-tags, lifting) mem-Collect-eq)

definition *r-amalgamation* ≡ Cn 3 *r-pdec2* [*r-parallel*]

lemma *r-amalgamation-recfn*: *recfn* 3 *r-amalgamation*
unfolding *r-amalgamation-def* **by** *simp*

lemma *r-amalgamation*: *eval r-amalgamation* [*i, j, x*] = *amalgamation i j x*

proof (cases parallel *i j x* ↑)

case True

then have *eval r-parallel* [*i, j, x*] ↑

by (*simp add: r-parallel'*)

then have *eval r-amalgamation* [*i, j, x*] ↑

unfolding *r-amalgamation-def* **by** *simp*

moreover from True have *amalgamation i j x* ↑

using *amalgamation-def* **by** *simp*

ultimately show ?thesis **by** *simp*

next

case False

then have *eval r-parallel* [*i, j, x*] ↓

by (*simp add: r-parallel'*)

then have *eval r-amalgamation* [*i, j, x*] = *eval r-pdec2* [the (*eval r-parallel* [*i, j, x*])]

unfolding *r-amalgamation-def* **by** *simp*

also have ... ↓ = *pdec2* (the (*eval r-parallel* [*i, j, x*]))

by *simp*

finally show ?thesis **by** (*simp add: False amalgamation-def r-parallel'*)

qed

The function *amalgamate* computes Gödel numbers of amalgamations. It corresponds to the function *a* from the proof sketch.

definition *amalgamate* :: nat ⇒ nat ⇒ nat **where**
amalgamate i j ≡ *smn* 1 (*encode r-amalgamation*) [*i, j*]

lemma *amalgamate*: φ (*amalgamate i j*) = *amalgamation i j*

proof

fix *x*

have φ (*amalgamate i j*) *x* = *eval r-phi* [*amalgamate i j, x*]

by (*simp add: phi-def*)

also have ... = *eval r-phi* [*smn* 1 (*encode r-amalgamation*) [*i, j*], *x*]

using *amalgamate-def* **by** *simp*

```

also have ... = eval r-phi
  [encode (Cn 1 (r-universal 3))
   (r-constn 0 (encode r-amalgamation) # map (r-constn 0) [i, j] @ map (Id 1) [0]), x]
using smn[of 1 encode r-amalgamation [i, j]] by (simp add: numeral-3-eq-3)
also have ... = eval r-phi
  [encode (Cn 1 (r-universal 3))
   (r-const (encode r-amalgamation) # [r-const i, r-const j, Id 1 0]), x]
  (is ... = eval r-phi [encode ?f, x])
by (simp add: r-constn-def)
finally have  $\varphi$  (amalgamate i j) x = eval r-phi
  [encode (Cn 1 (r-universal 3))
   (r-const (encode r-amalgamation) # [r-const i, r-const j, Id 1 0]), x] .
then have  $\varphi$  (amalgamate i j) x = eval (r-universal 3) [encode r-amalgamation, i, j, x]
  unfolding r-phi-def using r-universal[of ?f 1] r-amalgamation-recfn by simp
then show  $\varphi$  (amalgamate i j) x = amalgamation i j x
  using r-amalgamation by (simp add: r-amalgamation-recfn r-universal)
qed

```

```

lemma amalgamation-in-P1: amalgamation i j  $\in \mathcal{P}$ 
  using amalgamate by (metis P2-proj-P1 phi-in-P2)

```

```

lemma amalgamation-V01-R1:
  assumes  $f \in V_{01}$ 
  shows amalgamation (the (f 0)) (the (f 1))  $\in \mathcal{R}$ 
  using assms amalgamation-V01-total amalgamation-in-P1
  by (simp add: P1-total-imp-R1)

```

```

definition r-amalgamate  $\equiv$ 
  Cn 2 (r-smn 1 2) [r-dummy 1 (r-const (encode r-amalgamation)), Id 2 0, Id 2 1]

```

```

lemma r-amalgamate-recfn: recfn 2 r-amalgamate
  unfolding r-amalgamate-def by simp

```

```

lemma r-amalgamate: eval r-amalgamate [i, j]  $\downarrow =$  amalgamate i j

```

```

proof –
  let ?p = encode r-amalgamation
  have rs21: eval (r-smn 1 2) [?p, i, j]  $\downarrow =$  smn 1 ?p [i, j]
    using r-smn by simp
  moreover have eval r-amalgamate [i, j] = eval (r-smn 1 2) [?p, i, j]
    unfolding r-amalgamate-def by auto
  ultimately have eval r-amalgamate [i, j]  $\downarrow =$  smn 1 ?p [i, j]
    by simp
  then show ?thesis using amalgamate-def by simp
qed

```

The strategy S distinguishes the two cases from the proof sketch with the help of the next function, which checks if a hypothesis φ_i is inconsistent with a prefix e . If so, it returns the least $x < |e|$ witnessing the inconsistency; otherwise it returns the length $|e|$. If φ_i diverges for some $x < |e|$, so does the function.

```

definition inconsist :: partial2 where
  inconsist i e  $\equiv$ 
    (if  $\exists x < e\text{-length } e. \varphi i x \uparrow$  then None
     else if  $\exists x < e\text{-length } e. \varphi i x \downarrow \neq e\text{-nth } e x$ 
      then Some (LEAST  $x. x < e\text{-length } e \wedge \varphi i x \downarrow \neq e\text{-nth } e x$ )
      else Some (e-length e))

```

lemma *inconsist-converg*:

assumes *inconsist i e* \downarrow

shows *inconsist i e* =

(if $\exists x < e\text{-length } e. \varphi i x \downarrow \neq e\text{-nth } e x$

then *Some* (*LEAST* $x. x < e\text{-length } e \wedge \varphi i x \downarrow \neq e\text{-nth } e x$)

else *Some* (*e-length e*))

and $\forall x < e\text{-length } e. \varphi i x \downarrow$

using *inconsist-def assms* **by** (*presburger, meson*)

lemma *inconsist-bounded*:

assumes *inconsist i e* \downarrow

shows the (*inconsist i e*) $\leq e\text{-length } e$

proof (*cases* $\exists x < e\text{-length } e. \varphi i x \downarrow \neq e\text{-nth } e x$)

case *True*

then show *?thesis*

using *inconsist-converg*[*OF assms*]

by (*smt Least-le dual-order.strict-implies-order dual-order.strict-trans2 option.sel*)

next

case *False*

then show *?thesis* **using** *inconsist-converg*[*OF assms*] **by** *auto*

qed

lemma *inconsist-consistent*:

assumes *inconsist i e* \downarrow

shows *inconsist i e* $\downarrow = e\text{-length } e \longleftrightarrow (\forall x < e\text{-length } e. \varphi i x \downarrow = e\text{-nth } e x)$

proof

show $\forall x < e\text{-length } e. \varphi i x \downarrow = e\text{-nth } e x$ **if** *inconsist i e* $\downarrow = e\text{-length } e$

proof (*cases* $\exists x < e\text{-length } e. \varphi i x \downarrow \neq e\text{-nth } e x$)

case *True*

then show *?thesis*

using *that inconsist-converg*[*OF assms*]

by (*metis (mono-tags, lifting) not-less-Least option.inject*)

next

case *False*

then show *?thesis*

using *that inconsist-converg*[*OF assms*] **by** *simp*

qed

show $\forall x < e\text{-length } e. \varphi i x \downarrow = e\text{-nth } e x \implies \textit{inconsist i e} \downarrow = e\text{-length } e$

unfolding *inconsist-def* **using** *assms* **by** *auto*

qed

lemma *inconsist-converg-eq*:

assumes *inconsist i e* $\downarrow = e\text{-length } e$

shows $\forall x < e\text{-length } e. \varphi i x \downarrow = e\text{-nth } e x$

using *assms inconsist-consistent* **by** *auto*

lemma *inconsist-converg-less*:

assumes *inconsist i e* \downarrow **and** the (*inconsist i e*) $< e\text{-length } e$

shows $\exists x < e\text{-length } e. \varphi i x \downarrow \neq e\text{-nth } e x$

and *inconsist i e* $\downarrow = (\textit{LEAST } x. x < e\text{-length } e \wedge \varphi i x \downarrow \neq e\text{-nth } e x)$

proof –

show $\exists x < e\text{-length } e. \varphi i x \downarrow \neq e\text{-nth } e x$

using *assms* **by** (*metis (no-types, lifting) inconsist-converg(1) nat-neq-iff option.sel*)

then show *inconsist i e* $\downarrow = (\textit{LEAST } x. x < e\text{-length } e \wedge \varphi i x \downarrow \neq e\text{-nth } e x)$

using *assms inconsist-converg* **by** *presburger*

qed

lemma *least-bounded-Suc*:

assumes $\exists x. x < upper \wedge P x$

shows $(LEAST x. x < upper \wedge P x) = (LEAST x. x < Suc upper \wedge P x)$

proof –

let $?Q = \lambda x. x < upper \wedge P x$

let $?x = Least ?Q$

from *assms* have $?x < upper \wedge P ?x$

using *LeastI-ex*[of $?Q$] by *simp*

then have $1: ?x < Suc upper \wedge P ?x$ by *simp*

from *assms* have $2: \forall y < ?x. \neg P y$

using *Least-le*[of $?Q$] *not-less-Least* by *fastforce*

have $(LEAST x. x < Suc upper \wedge P x) = ?x$

proof (rule *Least-equality*)

show $?x < Suc upper \wedge P ?x$ using $1\ 2$ by *blast*

show $\bigwedge y. y < Suc upper \wedge P y \implies ?x \leq y$

using $1\ 2$ *leI* by *blast*

qed

then show *thesis* ..

qed

lemma *least-bounded-gr*:

fixes $P :: nat \Rightarrow bool$ and $m :: nat$

assumes $\exists x. x < upper \wedge P x$

shows $(LEAST x. x < upper \wedge P x) = (LEAST x. x < upper + m \wedge P x)$

proof (induction m)

case 0

then show *case* by *simp*

next

case $(Suc\ m)$

moreover have $\exists x. x < upper + m \wedge P x$

using *assms trans-less-add1* by *blast*

ultimately show *case* using *least-bounded-Suc* by *simp*

qed

lemma *inconsist-init-converg-less*:

assumes $f \in \mathcal{R}$

and $\varphi\ i \in \mathcal{R}$

and *inconsist* $i\ (f \triangleright n) \downarrow$

and *the* $(\text{inconsist } i\ (f \triangleright n)) < Suc\ n$

shows *inconsist* $i\ (f \triangleright (n + m)) = \text{inconsist } i\ (f \triangleright n)$

proof –

have *phi-i-total*: $\varphi\ i\ x \downarrow$ for x

using *assms* by *simp*

moreover have *f-nth*: $f\ x \downarrow = e\text{-nth}\ (f \triangleright n)\ x$ if $x < Suc\ n$ for $x\ n$

using *that assms(1)* by *simp*

ultimately have $(\varphi\ i\ x \neq f\ x) = (\varphi\ i\ x \downarrow \neq e\text{-nth}\ (f \triangleright n)\ x)$ if $x < Suc\ n$ for $x\ n$

using *that* by *simp*

then have *cond*: $(x < Suc\ n \wedge \varphi\ i\ x \neq f\ x) =$

$(x < e\text{-length}\ (f \triangleright n) \wedge \varphi\ i\ x \downarrow \neq e\text{-nth}\ (f \triangleright n)\ x)$ for $x\ n$

using *length-init* by *metis*

then have

$1: \exists x < Suc\ n. \varphi\ i\ x \neq f\ x$ and

$2: \text{inconsist } i\ (f \triangleright n) \downarrow = (LEAST\ x. x < Suc\ n \wedge \varphi\ i\ x \neq f\ x)$

using *assms(3,4)* *inconsist-converg-less*[of $i\ f \triangleright n$] by *simp-all*

then have 3: $\exists x < \text{Suc } (n + m). \varphi i x \neq f x$
using *not-add-less1* **by** *fastforce*
then have $\exists x < \text{Suc } (n + m). \varphi i x \downarrow \neq e\text{-nth } (f \triangleright (n + m)) x$
using *cond* **by** *blast*
then have $\exists x < e\text{-length } (f \triangleright (n + m)). \varphi i x \downarrow \neq e\text{-nth } (f \triangleright (n + m)) x$
by *simp*
moreover have 4: $\text{inconsist } i (f \triangleright (n + m)) \downarrow$
using *assms(2) R1-imp-total1 inconsist-def* **by** *simp*
ultimately have $\text{inconsist } i (f \triangleright (n + m)) \downarrow =$
 $(\text{LEAST } x. x < e\text{-length } (f \triangleright (n + m)) \wedge \varphi i x \downarrow \neq e\text{-nth } (f \triangleright (n + m)) x)$
using *inconsist-converg[OF 4]* **by** *simp*
then have 5: $\text{inconsist } i (f \triangleright (n + m)) \downarrow = (\text{LEAST } x. x < \text{Suc } (n + m) \wedge \varphi i x \neq f x)$
using *cond[of - n + m]* **by** *simp*
then have $(\text{LEAST } x. x < \text{Suc } n \wedge \varphi i x \neq f x) =$
 $(\text{LEAST } x. x < \text{Suc } n + m \wedge \varphi i x \neq f x)$
using *least-bounded-gr[where ?upper=Suc n] 1 3* **by** *simp*
then show *?thesis* **using** 2 5 **by** *simp*
qed

definition *r-inconsist* \equiv

let

$f = \text{Cn } 2 \text{ r-length } [\text{Id } 2 \ 1];$

$g = \text{Cn } 4 \text{ r-iffless}$

$[\text{Id } 4 \ 1,$

$\text{Cn } 4 \text{ r-length } [\text{Id } 4 \ 3],$

$\text{Id } 4 \ 1,$

$\text{Cn } 4 \text{ r-ifeq}$

$[\text{Cn } 4 \text{ r-phi } [\text{Id } 4 \ 2, \text{Id } 4 \ 0],$

$\text{Cn } 4 \text{ r-nth } [\text{Id } 4 \ 3, \text{Id } 4 \ 0],$

$\text{Id } 4 \ 1,$

$\text{Id } 4 \ 0]]$

in $\text{Cn } 2 (\text{Pr } 2 \ f \ g) [\text{Cn } 2 \ \text{r-length } [\text{Id } 2 \ 1], \text{Id } 2 \ 0, \text{Id } 2 \ 1]$

lemma *r-inconsist-recfn*: *recfn* 2 *r-inconsist*

unfolding *r-inconsist-def* **by** *simp*

lemma *r-inconsist*: *eval* *r-inconsist* $[i, e] = \text{inconsist } i \ e$

proof –

define *f* **where** $f = \text{Cn } 2 \ \text{r-length } [\text{Id } 2 \ 1]$

define *len* **where** $\text{len} = \text{Cn } 4 \ \text{r-length } [\text{Id } 4 \ 3]$

define *nth* **where** $\text{nth} = \text{Cn } 4 \ \text{r-nth } [\text{Id } 4 \ 3, \text{Id } 4 \ 0]$

define *ph* **where** $\text{ph} = \text{Cn } 4 \ \text{r-phi } [\text{Id } 4 \ 2, \text{Id } 4 \ 0]$

define *g* **where**

$g = \text{Cn } 4 \ \text{r-iffless } [\text{Id } 4 \ 1, \text{len}, \text{Id } 4 \ 1, \text{Cn } 4 \ \text{r-ifeq } [\text{ph}, \text{nth}, \text{Id } 4 \ 1, \text{Id } 4 \ 0]]$

have *recfn* 2 *f*

unfolding *f-def* **by** *simp*

have *f*: *eval* *f* $[i, e] \downarrow = e\text{-length } e$

unfolding *f-def* **by** *simp*

have *recfn* 4 *len*

unfolding *len-def* **by** *simp*

have *len*: *eval* *len* $[j, v, i, e] \downarrow = e\text{-length } e$ **for** *j v*

unfolding *len-def* **by** *simp*

have *recfn* 4 *nth*

unfolding *nth-def* **by** *simp*

have *nth*: *eval* *nth* $[j, v, i, e] \downarrow = e\text{-nth } e \ j$ **for** *j v*

unfolding *nth-def* **by** *simp*

```

have recfn 4 ph
  unfolding ph-def by simp
have ph: eval ph [j, v, i, e] =  $\varphi$  i j for j v
  unfolding ph-def using phi-def by simp
have recfn 4 g
  unfolding g-def using <recfn 4 nth> <recfn 4 ph> <recfn 4 len> by simp
have g-diverg: eval g [j, v, i, e]  $\uparrow$  if eval ph [j, v, i, e]  $\uparrow$  for j v
  unfolding g-def using that <recfn 4 nth> <recfn 4 ph> <recfn 4 len> by simp
have g-converg: eval g [j, v, i, e]  $\downarrow$  =
  (if v < e-length e then v else if  $\varphi$  i j  $\downarrow$  = e-nth e j then v else j)
  if eval ph [j, v, i, e]  $\downarrow$  for j v
  unfolding g-def using that <recfn 4 nth> <recfn 4 ph> <recfn 4 len> len nth ph
  by auto
define h where h  $\equiv$  Pr 2 f g
then have recfn 3 h
  by (simp add: <recfn 2 f> <recfn 4 g>)

let ?invariant =  $\lambda$  j i e.
  (if  $\exists x < j. \varphi$  i x  $\uparrow$  then None
   else if  $\exists x < j. \varphi$  i x  $\downarrow \neq$  e-nth e x
    then Some (LEAST x. x < j  $\wedge$   $\varphi$  i x  $\downarrow \neq$  e-nth e x)
   else Some (e-length e))

have eval h [j, i, e] = ?invariant j i e if j  $\leq$  e-length e for j
  using that
proof (induction j)
  case 0
  then show ?case unfolding h-def using <recfn 2 f> f <recfn 4 g> by simp
next
  case (Suc j)
  then have j-less: j < e-length e by simp
  then have j-le: j  $\leq$  e-length e by simp
  show ?case
  proof (cases eval h [j, i, e]  $\uparrow$ )
    case True
    then have  $\exists x < j. \varphi$  i x  $\uparrow$ 
      using j-le Suc.IH by (metis option.simps(3))
    then have  $\exists x < \text{Suc } j. \varphi$  i x  $\uparrow$ 
      using less-SucI by blast
    moreover have h: eval h [Suc j, i, e]  $\uparrow$ 
      using True h-def <recfn 3 h> by simp
    ultimately show ?thesis by simp
  next
    case False
    with Suc.IH j-le have h-j: eval h [j, i, e] =
      (if  $\exists x < j. \varphi$  i x  $\downarrow \neq$  e-nth e x
       then Some (LEAST x. x < j  $\wedge$   $\varphi$  i x  $\downarrow \neq$  e-nth e x)
       else Some (e-length e))
      by presburger
    then have the-h-j: the (eval h [j, i, e]) =
      (if  $\exists x < j. \varphi$  i x  $\downarrow \neq$  e-nth e x
       then LEAST x. x < j  $\wedge$   $\varphi$  i x  $\downarrow \neq$  e-nth e x
       else e-length e)
      (is - = ?v)
      by auto
    have h-Suc: eval h [Suc j, i, e] = eval g [j, the (eval h [j, i, e]), i, e]

```

```

using False h-def ‹recfn 4 g ‹recfn 2 f› by auto
show ?thesis
proof (cases  $\varphi\ i\ j\ \uparrow$ )
  case True
    with ph g-diverg h-Suc show ?thesis by auto
  next
    case False
      with h-Suc have eval h [Suc j, i, e] ↓=
        (if  $?v < e\text{-length}\ e$  then  $?v$ 
         else if  $\varphi\ i\ j\ \downarrow = e\text{-nth}\ e\ j$  then  $?v\ \text{else}\ j$ )
        (is -  $\downarrow = ?lhs$ )
      using g-converg ph the-h-j by simp
      moreover have ?invariant (Suc j) i e ↓=
        (if  $\exists x < Suc\ j.\ \varphi\ i\ x\ \downarrow \neq e\text{-nth}\ e\ x$ 
         then LEAST  $x.\ x < Suc\ j \wedge \varphi\ i\ x\ \downarrow \neq e\text{-nth}\ e\ x$ 
         else  $e\text{-length}\ e$ )
        (is -  $\downarrow = ?rhs$ )
      proof -
        from False have  $\varphi\ i\ j\ \downarrow$  by simp
        moreover have  $\neg (\exists x < j.\ \varphi\ i\ x\ \uparrow)$ 
          by (metis (no-types, lifting) Suc.IH h-j j-le option.simps(3))
        ultimately have  $\neg (\exists x < Suc\ j.\ \varphi\ i\ x\ \uparrow)$ 
          using less-Suc-eq by auto
        then show ?thesis by auto
      qed
      moreover have  $?lhs = ?rhs$ 
      proof (cases  $?v < e\text{-length}\ e$ )
        case True
          then have
            ex-j:  $\exists x < j.\ \varphi\ i\ x\ \downarrow \neq e\text{-nth}\ e\ x$  and
            v-eq:  $?v = (\text{LEAST}\ x.\ x < j \wedge \varphi\ i\ x\ \downarrow \neq e\text{-nth}\ e\ x)$ 
            by presburger+
          with True have  $?lhs = ?v$  by simp
          from ex-j have  $\exists x < Suc\ j.\ \varphi\ i\ x\ \downarrow \neq e\text{-nth}\ e\ x$ 
            using less-SucI by blast
          then have  $?rhs = (\text{LEAST}\ x.\ x < Suc\ j \wedge \varphi\ i\ x\ \downarrow \neq e\text{-nth}\ e\ x)$  by simp
          with True v-eq ex-j show ?thesis
            using least-bounded-Suc[of j λx. φ i x ↓ ≠ e-nth e x] by simp
        next
          case False
            then have not-ex:  $\neg (\exists x < j.\ \varphi\ i\ x\ \downarrow \neq e\text{-nth}\ e\ x)$ 
              using Least-le[of λx. x < j ∧ φ i x ↓ ≠ e-nth e x] j-le
              by (smt leD le-less-linear le-trans)
            then have  $?v = e\text{-length}\ e$  by argo
            with False have lhs:  $?lhs = (\text{if}\ \varphi\ i\ j\ \downarrow = e\text{-nth}\ e\ j\ \text{then}\ e\text{-length}\ e\ \text{else}\ j)$ 
              by simp
            show ?thesis
            proof (cases  $\varphi\ i\ j\ \downarrow = e\text{-nth}\ e\ j$ )
              case True
                then have  $\neg (\exists x < Suc\ j.\ \varphi\ i\ x\ \downarrow \neq e\text{-nth}\ e\ x)$ 
                  using less-SucE not-ex by blast
                then have  $?rhs = e\text{-length}\ e$  by argo
                moreover from True have  $?lhs = e\text{-length}\ e$ 
                  using lhs by simp
                ultimately show ?thesis by simp
              next case False

```

```

then have  $\varphi i j \downarrow \neq e\text{-nth } e j$ 
  using  $\langle \varphi i j \downarrow \rangle$  by simp
with not-ex have (LEAST  $x. x < \text{Suc } j \wedge \varphi i x \downarrow \neq e\text{-nth } e x$ ) =  $j$ 
  using LeastI[of  $\lambda x. x < \text{Suc } j \wedge \varphi i x \downarrow \neq e\text{-nth } e x j$ ] less-Suc-eq
  by blast
then have  $?rhs = j$ 
  using  $\langle \varphi i j \downarrow \neq e\text{-nth } e j \rangle$  by (meson lessI)
moreover from False lhs have  $?lhs = j$  by simp
ultimately show  $?thesis$  by simp
qed
qed
ultimately show  $?thesis$  by simp
qed
qed
qed
then have  $\text{eval } h [e\text{-length } e, i, e] = ?\text{invariant } (e\text{-length } e) i e$ 
  by auto
then have  $\text{eval } h [e\text{-length } e, i, e] = \text{inconsist } i e$ 
  using inconsist-def by simp
moreover have  $\text{eval } (Cn 2 (Pr 2 f g) [Cn 2 r\text{-length } [Id 2 1], Id 2 0, Id 2 1]) [i, e] =$ 
   $\text{eval } h [e\text{-length } e, i, e]$ 
  using  $\langle \text{recfn } 4 g \rangle \langle \text{recfn } 2 f \rangle h\text{-def}$  by auto
ultimately show  $?thesis$ 
  unfolding  $r\text{-inconsist-def}$  by (simp add:  $f\text{-def } g\text{-def } len\text{-def } nth\text{-def } ph\text{-def}$ )
qed

```

lemma *inconsist-for-total*:

```

assumes total1 ( $\varphi i$ )
shows  $\text{inconsist } i e \downarrow =$ 
  (if  $\exists x < e\text{-length } e. \varphi i x \downarrow \neq e\text{-nth } e x$ 
    then LEAST  $x. x < e\text{-length } e \wedge \varphi i x \downarrow \neq e\text{-nth } e x$ 
    else  $e\text{-length } e$ )
unfolding inconsist-def using assms total1-def by (auto; blast)

```

lemma *inconsist-for-V01*:

```

assumes  $f \in V_{01}$  and  $k = \text{amalgamate } (the (f 0)) (the (f 1))$ 
shows  $\text{inconsist } k e \downarrow =$ 
  (if  $\exists x < e\text{-length } e. \varphi k x \downarrow \neq e\text{-nth } e x$ 
    then LEAST  $x. x < e\text{-length } e \wedge \varphi k x \downarrow \neq e\text{-nth } e x$ 
    else  $e\text{-length } e$ )

```

proof –

```

have  $\varphi k \in \mathcal{R}$ 
  using amalgamation-V01-R1[OF assms(1)] assms(2) amalgamate by simp
then have total1 ( $\varphi k$ ) by simp
with inconsist-for-total[of  $k$ ] show  $?thesis$  by simp

```

qed

The next function computes Gödel numbers of functions consistent with a given prefix. The strategy will use these as consistent auxiliary hypotheses when receiving a prefix of length one.

definition $r\text{-auxhyp} \equiv Cn 1 (r\text{-smn } 1 1) [r\text{-const } (\text{encode } r\text{-prenum}), Id 1 0]$

lemma *r-auxhyp-prim*: $\text{prim-recfn } 1 r\text{-auxhyp}$

unfolding $r\text{-auxhyp-def}$ by simp

lemma *r-auxhyp*: $\varphi (the (\text{eval } r\text{-auxhyp } [e])) = \text{prenum } e$

```

proof
  fix  $x$ 
  let  $?p = \text{encode } r\text{-prenum}$ 
  let  $?p = \text{encode } r\text{-prenum}$ 
  have  $\text{eval } r\text{-auxhyp } [e] = \text{eval } (r\text{-smn } 1 \ 1) \ [?p, e]$ 
    unfolding  $r\text{-auxhyp-def}$  by  $\text{simp}$ 
  then have  $\text{eval } r\text{-auxhyp } [e] \downarrow = \text{smn } 1 \ ?p \ [e]$ 
    by  $(\text{simp add: } r\text{-smn})$ 
  also have  $\dots \downarrow = \text{encode } (Cn \ 1 \ (r\text{-universal } (1 + \text{length } [e])))$ 
     $(r\text{-constn } (1 - 1) \ ?p \ \#$ 
     $\text{map } (r\text{-constn } (1 - 1)) \ [e] \ @ \ \text{map } (\text{recf.Id } 1) \ [0..<1]))$ 
    using  $\text{smn}[of \ 1 \ ?p \ [e]]$  by  $\text{simp}$ 
  also have  $\dots \downarrow = \text{encode } (Cn \ 1 \ (r\text{-universal } (1 + 1)))$ 
     $(r\text{-constn } 0 \ ?p \ \# \ \text{map } (r\text{-constn } 0) \ [e] \ @ \ [Id \ 1 \ 0]))$ 
    by  $\text{simp}$ 
  also have  $\dots \downarrow = \text{encode } (Cn \ 1 \ (r\text{-universal } 2))$ 
     $(r\text{-constn } 0 \ ?p \ \# \ \text{map } (r\text{-constn } 0) \ [e] \ @ \ [Id \ 1 \ 0]))$ 
    by  $(\text{metis one-add-one})$ 
  also have  $\dots \downarrow = \text{encode } (Cn \ 1 \ (r\text{-universal } 2) \ [r\text{-constn } 0 \ ?p, r\text{-constn } 0 \ e, Id \ 1 \ 0])$ 
    by  $\text{simp}$ 
  also have  $\dots \downarrow = \text{encode } (Cn \ 1 \ (r\text{-universal } 2) \ [r\text{-const } ?p, r\text{-const } e, Id \ 1 \ 0])$ 
    using  $r\text{-constn-def}$  by  $\text{simp}$ 
  finally have  $\text{eval } r\text{-auxhyp } [e] \downarrow =$ 
     $\text{encode } (Cn \ 1 \ (r\text{-universal } 2) \ [r\text{-const } ?p, r\text{-const } e, Id \ 1 \ 0]) .$ 
  moreover have  $\varphi \ (the \ (\text{eval } r\text{-auxhyp } [e])) \ x = \text{eval } r\text{-phi } [the \ (\text{eval } r\text{-auxhyp } [e]), x]$ 
    by  $(\text{simp add: } phi\text{-def})$ 
  ultimately have  $\varphi \ (the \ (\text{eval } r\text{-auxhyp } [e])) \ x =$ 
     $\text{eval } r\text{-phi } [\text{encode } (Cn \ 1 \ (r\text{-universal } 2) \ [r\text{-const } ?p, r\text{-const } e, Id \ 1 \ 0]), x]$ 
     $(\text{is } - = \text{eval } r\text{-phi } [\text{encode } ?f, x])$ 
    by  $\text{simp}$ 
  then have  $\varphi \ (the \ (\text{eval } r\text{-auxhyp } [e])) \ x =$ 
     $\text{eval } (Cn \ 1 \ (r\text{-universal } 2) \ [r\text{-const } ?p, r\text{-const } e, Id \ 1 \ 0]) \ [x]$ 
    using  $r\text{-phi-def } r\text{-universal}[of \ ?f \ 1 \ [x]]$  by  $\text{simp}$ 
  then have  $\varphi \ (the \ (\text{eval } r\text{-auxhyp } [e])) \ x = \text{eval } (r\text{-universal } 2) \ [?p, e, x]$ 
    by  $\text{simp}$ 
  then have  $\varphi \ (the \ (\text{eval } r\text{-auxhyp } [e])) \ x = \text{eval } r\text{-prenum } [e, x]$ 
    using  $r\text{-universal}$  by  $\text{simp}$ 
  then show  $\varphi \ (the \ (\text{eval } r\text{-auxhyp } [e])) \ x = \text{prenum } e \ x$  by  $\text{simp}$ 
qed

```

definition $\text{auxhyp} :: \text{partial1}$ **where**

$\text{auxhyp } e \equiv \text{eval } r\text{-auxhyp } [e]$

lemma auxhyp-prenum : $\varphi \ (the \ (\text{auxhyp } e)) = \text{prenum } e$

using $\text{auxhyp-def } r\text{-auxhyp}$ **by** metis

lemma auxhyp-in-R1 : $\text{auxhyp} \in \mathcal{R}$

using $\text{auxhyp-def } Mn\text{-free-imp-total } R1I \ r\text{-auxhyp-prim}$ **by** metis

Now we can define our consistent learning strategy for V_{01} .

definition $r\text{-sv01} \equiv$

let

$at0 = Cn \ 1 \ r\text{-nth } [Id \ 1 \ 0, Z];$

$at1 = Cn \ 1 \ r\text{-nth } [Id \ 1 \ 0, r\text{-const } 1];$

$m = Cn \ 1 \ r\text{-amalgamate } [at0, at1];$

$c = Cn \ 1 \ r\text{-inconsist } [m, Id \ 1 \ 0];$

$p = Cn\ 1\ r\text{-pdec1}\ [Cn\ 1\ r\text{-parallel}\ [at0,\ at1,\ c]];$
 $g = Cn\ 1\ r\text{-ifeq}\ [c,\ r\text{-length},\ m,\ Cn\ 1\ r\text{-ifz}\ [p,\ at1,\ at0]]$
 $in\ Cn\ 1\ (r\text{-ifz}\ r\text{-auxhyp}\ g)\ [Cn\ 1\ r\text{-eq}\ [r\text{-length},\ r\text{-const}\ 1],\ Id\ 1\ 0]$

lemma *r-sv01-recfn*: *recfn 1 r-sv01*

unfolding *r-sv01-def* **using** *r-auxhyp-prim r-inconsist-recfn r-amalgamate-recfn*
by (*simp add: Let-def*)

definition *sv01* :: *partial1* ($\langle s_{01} \rangle$) **where**

sv01 *e* \equiv *eval r-sv01* [*e*]

lemma *sv01-in-P1*: $s_{01} \in \mathcal{P}$

using *sv01-def r-sv01-recfn P1I* **by** *presburger*

We are interested in the behavior of s_{01} only on prefixes of functions in V_{01} . This behavior is linked to the amalgamation of $f(0)$ and $f(1)$, where f is the function to be learned.

abbreviation *amalg01* :: *partial1* \Rightarrow *nat* **where**

amalg01 *f* \equiv *amalgamate* (*the* (*f* 0)) (*the* (*f* 1))

lemma *sv01*:

assumes $f \in V_{01}$

shows $s_{01}\ (f \triangleright 0) = \text{auxhyp}\ (f \triangleright 0)$

and $n \neq 0 \implies$

$\text{inconsist}\ (\text{amalg01}\ f)\ (f \triangleright n) \downarrow = \text{Suc}\ n \implies$

$s_{01}\ (f \triangleright n) \downarrow = \text{amalg01}\ f$

and $n \neq 0 \implies$

$\text{the}\ (\text{inconsist}\ (\text{amalg01}\ f)\ (f \triangleright n)) < \text{Suc}\ n \implies$

$\text{pdec1}\ (\text{the}\ (\text{parallel}\ (\text{the}\ (f\ 0))\ (\text{the}\ (f\ 1))\ (\text{the}\ (\text{inconsist}\ (\text{amalg01}\ f)\ (f \triangleright n)))) = 0 \implies$

$s_{01}\ (f \triangleright n) = f\ 1$

and $n \neq 0 \implies$

$\text{the}\ (\text{inconsist}\ (\text{amalg01}\ f)\ (f \triangleright n)) < \text{Suc}\ n \implies$

$\text{pdec1}\ (\text{the}\ (\text{parallel}\ (\text{the}\ (f\ 0))\ (\text{the}\ (f\ 1))\ (\text{the}\ (\text{inconsist}\ (\text{amalg01}\ f)\ (f \triangleright n)))) \neq 0 \implies$

$s_{01}\ (f \triangleright n) = f\ 0$

proof –

have *f-total*: $\bigwedge x. f\ x \downarrow$

using *assms V01-def R1-imp-total1* **by** *blast*

define *at0* **where** *at0* = *Cn 1 r-nth* [*Id 1 0*, *Z*]

define *at1* **where** *at1* = *Cn 1 r-nth* [*Id 1 0*, *r-const 1*]

define *m* **where** *m* = *Cn 1 r-amalgamate* [*at0*, *at1*]

define *c* **where** *c* = *Cn 1 r-inconsist* [*m*, *Id 1 0*]

define *p* **where** *p* = *Cn 1 r-pdec1* [*Cn 1 r-parallel* [*at0*, *at1*, *c*]]

define *g* **where** *g* = *Cn 1 r-ifeq* [*c*, *r-length*, *m*, *Cn 1 r-ifz* [*p*, *at1*, *at0*]]

have *recfn 1 g*

unfolding *g-def p-def c-def m-def at1-def at0-def*

using *r-auxhyp-prim r-inconsist-recfn r-amalgamate-recfn*

by *simp*

have *eval* (*Cn 1 r-eq* [*r-length*, *r-const 1*]) [*f* \triangleright 0] $\downarrow = 0$

by *simp*

then have *eval r-sv01* [*f* \triangleright 0] = *eval r-auxhyp* [*f* \triangleright 0]

unfolding *r-sv01-def* **using** $\langle \text{recfn } 1\ g \rangle$ *c-def g-def m-def p-def r-auxhyp-prim*

by (*auto simp add: Let-def*)

then show $s_{01}\ (f \triangleright 0) = \text{auxhyp}\ (f \triangleright 0)$

by (*simp add: auxhyp-def sv01-def*)

have *sv01*: $s_{01}\ (f \triangleright n) = \text{eval}\ g\ [f \triangleright n]$ **if** $n \neq 0$

proof –

have *: $eval (Cn\ 1\ r\text{-}eq\ [r\text{-}length,\ r\text{-}const\ 1])\ [f\ \triangleright\ n]\ \downarrow\neq\ 0$
 (is $?r\text{-}eq\ \downarrow\neq\ 0$)
using *that* **by** *simp*
moreover **have** $recfn\ 2\ (r\text{-}lifz\ r\text{-}auxhyp\ g)$
using $\langle recfn\ 1\ g \rangle\ r\text{-}auxhyp\text{-}prim$ **by** *simp*
moreover **have** $eval\ r\text{-}sv01\ [f\ \triangleright\ n] =$
 $eval\ (Cn\ 1\ (r\text{-}lifz\ r\text{-}auxhyp\ g)\ [Cn\ 1\ r\text{-}eq\ [r\text{-}length,\ r\text{-}const\ 1],\ Id\ 1\ 0])\ [f\ \triangleright\ n]$
using $r\text{-}sv01\text{-}def$ **by** (*metis* $at0\text{-}def\ at1\text{-}def\ c\text{-}def\ g\text{-}def\ m\text{-}def\ p\text{-}def$)
ultimately **have** $eval\ r\text{-}sv01\ [f\ \triangleright\ n] = eval\ (r\text{-}lifz\ r\text{-}auxhyp\ g)\ [the\ ?r\text{-}eq,\ f\ \triangleright\ n]$
by *simp*
then **have** $eval\ r\text{-}sv01\ [f\ \triangleright\ n] = eval\ g\ [f\ \triangleright\ n]$
using * $\langle recfn\ 1\ g \rangle\ r\text{-}auxhyp\text{-}prim$ **by** *auto*
then **show** $?thesis$ **by** (*simp* $add: sv01\text{-}def\ that$)
qed

have $recfn\ 1\ at0$
unfolding $at0\text{-}def$ **by** *simp*
have $at0: eval\ at0\ [f\ \triangleright\ n]\ \downarrow = the\ (f\ 0)$
unfolding $at0\text{-}def$ **by** *simp*
have $recfn\ 1\ at1$
unfolding $at1\text{-}def$ **by** *simp*
have $at1: n \neq 0 \implies eval\ at1\ [f\ \triangleright\ n]\ \downarrow = the\ (f\ 1)$
unfolding $at1\text{-}def$ **by** *simp*
have $recfn\ 1\ m$
unfolding $m\text{-}def\ at0\text{-}def\ at1\text{-}def$ **using** $r\text{-}amalgamate\text{-}recfn$ **by** *simp*
have $m: n \neq 0 \implies eval\ m\ [f\ \triangleright\ n]\ \downarrow = amalg01\ f$
 (is $- \implies - \downarrow = ?m$)
unfolding $m\text{-}def\ at0\text{-}def\ at1\text{-}def$
using $at0\ at1\ amalgamate\ r\text{-}amalgamate\ r\text{-}amalgamate\text{-}recfn$ **by** *simp*
then **have** $c: n \neq 0 \implies eval\ c\ [f\ \triangleright\ n] = inconsist\ (amalg01\ f)\ (f\ \triangleright\ n)$
 (is $- \implies - = ?c$)
unfolding $c\text{-}def$ **using** $r\text{-}inconsistent\text{-}recfn\ \langle recfn\ 1\ m \rangle\ r\text{-}inconsistent$ **by** *auto*
then **have** $c\text{-}converg: n \neq 0 \implies eval\ c\ [f\ \triangleright\ n]\ \downarrow$
using $inconsistent\text{-}for\ V01[OF\ assms]$ **by** *simp*
have $recfn\ 1\ c$
unfolding $c\text{-}def$ **using** $\langle recfn\ 1\ m \rangle\ r\text{-}inconsistent\text{-}recfn$ **by** *simp*

have $par: n \neq 0 \implies$
 $eval\ (Cn\ 1\ r\text{-}parallel\ [at0,\ at1,\ c])\ [f\ \triangleright\ n] = parallel\ (the\ (f\ 0))\ (the\ (f\ 1))\ (the\ ?c)$
 (is $- \implies - = ?par$)
using $at0\ at1\ c\ c\text{-}converg\ m\ r\text{-}parallel'$ $\langle recfn\ 1\ at0 \rangle\ \langle recfn\ 1\ at1 \rangle\ \langle recfn\ 1\ c \rangle$
by *simp*
with $parallel\text{-}converg\ V01[OF\ assms]$ **have**
 $par\text{-}converg: n \neq 0 \implies eval\ (Cn\ 1\ r\text{-}parallel\ [at0,\ at1,\ c])\ [f\ \triangleright\ n]\ \downarrow$
by *simp*
then **have** $p\text{-}converg: n \neq 0 \implies eval\ p\ [f\ \triangleright\ n]\ \downarrow$
unfolding $p\text{-}def$ **using** $at0\ at1\ c\text{-}converg\ \langle recfn\ 1\ at0 \rangle\ \langle recfn\ 1\ at1 \rangle\ \langle recfn\ 1\ c \rangle$
by *simp*
have $p: n \neq 0 \implies eval\ p\ [f\ \triangleright\ n]\ \downarrow = pdec1\ (the\ ?par)$
unfolding $p\text{-}def$
using $at0\ at1\ c\text{-}converg\ \langle recfn\ 1\ at0 \rangle\ \langle recfn\ 1\ at1 \rangle\ \langle recfn\ 1\ c \rangle\ par\ par\text{-}converg$
by *simp*
have $recfn\ 1\ p$
unfolding $p\text{-}def$ **using** $\langle recfn\ 1\ at0 \rangle\ \langle recfn\ 1\ at1 \rangle\ \langle recfn\ 1\ m \rangle\ \langle recfn\ 1\ c \rangle$
by *simp*

```

let ?r = Cn 1 r-ifz [p, at1, at0]
have r: n ≠ 0 ⇒ eval ?r [f ▷ n] = (if pdec1 (the ?par) = 0 then f 1 else f 0)
  using at0 at1 c-converg ⟨recfn 1 at0⟩ ⟨recfn 1 at1⟩ ⟨recfn 1 c⟩
  ⟨recfn 1 m⟩ ⟨recfn 1 p⟩ p f-total
  by fastforce

have g: n ≠ 0 ⇒
  eval g [f ▷ n] ↓=
    (if the ?c = e-length (f ▷ n)
     then ?m else the (eval (Cn 1 r-ifz [p, at1, at0]) [f ▷ n]))
  unfolding g-def
  using ⟨recfn 1 p⟩ ⟨recfn 1 at0⟩ ⟨recfn 1 at1⟩ ⟨recfn 1 c⟩ ⟨recfn 1 m⟩
  p-converg at1 at0 c c-converg m
  by simp
{
  assume n ≠ 0 and ?c ↓= Suc n
  moreover have e-length (f ▷ n) = Suc n by simp
  ultimately have eval g [f ▷ n] ↓= ?m using g by simp
  then show s01 (f ▷ n) ↓= amalg01 f
    using sv01[OF ⟨n ≠ 0⟩] by simp
next
  assume n ≠ 0 and the ?c < Suc n and pdec1 (the ?par) = 0
  with g r f-total have eval g [f ▷ n] = f 1 by simp
  then show s01 (f ▷ n) = f 1
    using sv01[OF ⟨n ≠ 0⟩] by simp
next
  assume n ≠ 0 and the ?c < Suc n and pdec1 (the ?par) ≠ 0
  with g r f-total have eval g [f ▷ n] = f 0 by simp
  then show s01 (f ▷ n) = f 0
    using sv01[OF ⟨n ≠ 0⟩] by simp
}
qed

```

Part of the correctness of s_{01} is convergence on prefixes of functions in V_{01} .

lemma *sv01-converg-V01*:

```

assumes f ∈ V01
shows s01 (f ▷ n) ↓
proof (cases n = 0)
  case True
  then show ?thesis
    using assms sv01 R1-imp-total1 aucthyp-in-R1 by simp
next
  case n-gr-0: False
  show ?thesis
  proof (cases inconsist (amalg01 f) (f ▷ n) ↓= Suc n)
    case True
    then show ?thesis
      using n-gr-0 assms sv01 by simp
  next
    case False
    then have the (inconsist (amalg01 f) (f ▷ n)) < Suc n
      using assms inconsist-bounded inconsist-for-V01 length-init
      by (metis (no-types, lifting) le-neq-implies-less option.collapse option.simps(3))
    then show ?thesis
      using n-gr-0 assms sv01 R1-imp-total1 total1E V01-def

```

```

    by (metis (no-types, lifting) mem-Collect-eq)
  qed
qed

```

Another part of the correctness of s_{01} is its hypotheses being consistent on prefixes of functions in V_{01} .

```

lemma sv01-consistent-V01:
  assumes f ∈ V01
  shows ∀ x ≤ n. φ (the (s01 (f ▷ n))) x = f x
proof (cases n = 0)
  case True
  then have s01 (f ▷ n) = auxhyp (f ▷ n)
    using sv01[OF assms] by simp
  then have φ (the (s01 (f ▷ n))) = prenum (f ▷ n)
    using auxhyp-prenum by simp
  then show ?thesis
    using R1-imp-total1 total1E assms by (simp add: V01-def)
next
  case n-gr-0: False
  let ?m = amalg01 f
  let ?e = f ▷ n
  let ?c = the (inconsist ?m ?e)
  have c: inconsist ?m ?e ↓
    using assms inconsist-for-V01 by blast
  show ?thesis
proof (cases inconsist ?m ?e ↓ = Suc n)
  case True
  then show ?thesis
    using assms n-gr-0 sv01 R1-imp-total1 total1E V01-def is-init-of-def
      inconsist-consistent not-initial-imp-not-eq length-init inconsist-converg-eq
      by (metis (no-types, lifting) le-imp-less-Suc mem-Collect-eq option.sel)
next
  case False
  then have less: the (inconsist ?m ?e) < Suc n
    using c assms inconsist-bounded inconsist-for-V01 length-init
    by (metis le-neq-implies-less option.collapse)
  then have the (inconsist ?m ?e) < e-length ?e
    by auto
  then have
    ∃ x < e-length ?e. φ ?m x ↓ ≠ e-nth ?e x
    inconsist ?m ?e ↓ = (LEAST x. x < e-length ?e ∧ φ ?m x ↓ ≠ e-nth ?e x)
    (is - ↓ = Least ?P)
    using inconsist-converg-less[OF c] by simp-all
  then have ?P ?c and ∧ x. x < ?c ⇒ ¬ ?P x
    using LeastI-ex[of ?P] not-less-Least[of - ?P] by (auto simp del: e-nth)
  then have φ ?m ?c ≠ f ?c by auto
  then have amalgamation (the (f 0)) (the (f 1)) ?c ≠ f ?c
    using amalgamate by simp
  then have *: Some (pdec2 (the (parallel (the (f 0)) (the (f 1)) ?c))) ≠ f ?c
    using amalgamation-def by (metis assms parallel-converg-V01)
  let ?p = parallel (the (f 0)) (the (f 1)) ?c
  show ?thesis
proof (cases pdec1 (the ?p) = 0)
  case True
  then have φ (the (f 0)) ?c ↓ = pdec2 (the ?p)
    using assms parallel-0 parallel-converg-V01

```

```

    by (metis option.collapse prod.collapse prod-decode-inverse)
  then have  $\varphi$  (the (f 0)) ?c  $\neq$  f ?c
    using * by simp
  then have  $\varphi$  (the (f 0))  $\neq$  f by auto
  then have  $\varphi$  (the (f 1)) = f
    using assms V01-def by auto
  moreover have  $s_{01}$  (f  $\triangleright$  n) = f 1
    using True less n-gr-0 sv01 assms by simp
  ultimately show ?thesis by simp
next
case False
then have pdec1 (the ?p) = 1
  by (meson assms parallel-converg-V01 parallel-converg-pdec1-0-or-1)
then have  $\varphi$  (the (f 1)) ?c  $\downarrow$ = pdec2 (the ?p)
  using assms parallel-1 parallel-converg-V01
  by (metis option.collapse prod.collapse prod-decode-inverse)
then have  $\varphi$  (the (f 1)) ?c  $\neq$  f ?c
  using * by simp
then have  $\varphi$  (the (f 1))  $\neq$  f by auto
then have  $\varphi$  (the (f 0)) = f
  using assms V01-def by auto
moreover from False less n-gr-0 sv01 assms have  $s_{01}$  (f  $\triangleright$  n) = f 0
  by simp
ultimately show ?thesis by simp
qed
qed
qed

```

The final part of the correctness is s_{01} converging for all functions in V_{01} .

lemma *sv01-limit-V01*:

```

assumes  $f \in V_{01}$ 
shows  $\exists i. \forall^\infty n. s_{01}$  (f  $\triangleright$  n)  $\downarrow$ = i
proof (cases  $\forall n > 0. s_{01}$  (f  $\triangleright$  n)  $\downarrow$ = amalgamate (the (f 0)) (the (f 1)))
  case True
  then show ?thesis by (meson less-le-trans zero-less-one)
next
case False
then obtain  $n_0$  where  $n_0$ :
   $n_0 \neq 0$ 
   $s_{01}$  (f  $\triangleright$   $n_0$ )  $\downarrow \neq$  amalg01 f
  using  $\langle f \in V_{01} \rangle$  sv01-converg-V01 by blast
then have *: the (inconsist (amalg01 f) (f  $\triangleright$   $n_0$ )) < Suc  $n_0$ 
  (is the (inconsist ?m (f  $\triangleright$   $n_0$ )) < Suc  $n_0$ )
  using assms  $\langle n_0 \neq 0 \rangle$  sv01(2) inconsist-bounded inconsist-for-V01 length-init
  by (metis (no-types, lifting) le-neq-implies-less option.collapse option.simps(3))
moreover have  $f \in \mathcal{R}$ 
  using assms V01-def by auto
moreover have  $\varphi$  ?m  $\in \mathcal{R}$ 
  using amalgamate amalgamation-V01-R1 assms by auto
moreover have inconsist ?m (f  $\triangleright$   $n_0$ )  $\downarrow$ 
  using inconsist-for-V01 assms by blast
ultimately have **: inconsist ?m (f  $\triangleright$  ( $n_0$  + m)) = inconsist ?m (f  $\triangleright$   $n_0$ ) for m
  using inconsist-init-converg-less[of f ?m] by simp
then have the (inconsist ?m (f  $\triangleright$  ( $n_0$  + m))) < Suc  $n_0$  + m for m
  using * by auto
moreover have

```

```

pdec1 (the (parallel (the (f 0)) (the (f 1)) (the (inconsist ?m (f ▷ (n0 + m)))))) =
  pdec1 (the (parallel (the (f 0)) (the (f 1)) (the (inconsist ?m (f ▷ n0))))
for m
using ** by auto
moreover have n0 + m ≠ 0 for m
  using ⟨n0 ≠ 0⟩ by simp
ultimately have s01 (f ▷ (n0 + m)) = s01 (f ▷ n0) for m
  using assms sv01 * ⟨n0 ≠ 0⟩ by (metis add-Suc)
moreover define i where i = s01 (f ▷ n0)
ultimately have ∀ n ≥ n0. s01 (f ▷ n) = i
  using nat-le-iff-add by auto
then have ∀ n ≥ n0. s01 (f ▷ n) ↓ = the i
  using n0(2) by simp
then show ?thesis by auto
qed

```

```

lemma V01-learn-cons: learn-cons φ V01 s01
proof (rule learn-consI2)
  show environment φ V01 s01
    by (simp add: Collect-mono V01-def phi-in-P2 sv01-in-P1 sv01-converg-V01)
  show ∧ f n. f ∈ V01 ⇒ ∀ k ≤ n. φ (the (s01 (f ▷ n))) k = f k
    using sv01-consistent-V01 .
  show ∃ i n0. ∀ n ≥ n0. s01 (f ▷ n) ↓ = i if f ∈ V01 for f
    using sv01-limit-V01 that by simp
qed

```

```

corollary V01-in-CONS: V01 ∈ CONS
using V01-learn-cons CONS-def by auto

```

Now we can show the main result of this section, namely that there is a consistently learnable class that cannot be learned consistently by a total strategy. In other words, there is no Lemma R for CONS.

```

lemma no-lemma-R-for-CONS: ∃ U. U ∈ CONS ∧ (¬ (∃ s. s ∈ ℛ ∧ learn-cons φ U s))
using V01-in-CONS V01-not-in-R-cons by auto

```

end

2.9 LIM is a proper subset of BC

```

theory LIM-BC
imports Lemma-R
begin

```

The proper inclusion of LIM in BC has been proved by Barzdin [2] (see also Case and Smith [6]). The proof constructs a class $V \in \text{BC} - \text{LIM}$ by diagonalization against all LIM strategies. Exploiting Lemma R for LIM, we can assume that all such strategies are total functions. From the effective version of this lemma we derive a numbering $\sigma \in \mathcal{R}^2$ such that for all $U \in \text{LIM}$ there is an i with $U \in \text{LIM}_\varphi(\sigma_i)$. The idea behind V is for every i to construct a class V_i of cardinality one or two such that $V_i \notin \text{LIM}_\varphi(\sigma_i)$. It then follows that the union $V := \bigcup_i V_i$ cannot be learned by any σ_i and thus $V \notin \text{LIM}$. At the same time, the construction ensures that the functions in V are “predictable enough” to be learnable in the BC sense.

At the core is a process that maintains a state (b, k) of a list b of numbers and an index $k < |b|$ into this list. We imagine b to be the prefix of the function being constructed,

except for position k where we imagine b to have a “gap”; that is, b_k is not defined yet. Technically, we will always have $b_k = 0$, so b also represents the prefix after the “gap is filled” with 0, whereas $b_{k:=1}$ represents the prefix where the gap is filled with 1. For every $i \in \mathbb{N}$, the process starts in state $(i0, 1)$ and computes the next state from a given state (b, k) as follows:

1. if $\sigma_i(b_{<k}) \neq \sigma_i(b)$ then the next state is $(b0, |b|)$,
2. else if $\sigma_i(b_{<k}) \neq \sigma_i(b_{k:=1})$ then the next state is $(b_{k:=1}0, |b|)$,
3. else the next state is $(b0, k)$.

In other words, if σ_i changes its hypothesis when the gap in b is filled with 0 or 1, then the process fills the gap with 0 or 1, respectively, and appends a gap to b . If, however, a hypothesis change cannot be enforced at this point, the process appends a 0 to b and leaves the gap alone. Now there are two cases:

- Case 1. Every gap gets filled eventually. Then the process generates increasing prefixes of a total function τ_i , on which σ_i changes its hypothesis infinitely often. We set $V_i := \{\tau_i\}$, and have $V_i \notin \text{LIM}_\varphi(\sigma_i)$.
- Case 2. Some gap never gets filled. That means a state (b, k) is reached such that $\sigma_i(b0^t) = \sigma_i(b_{k:=1}0^t) = \sigma_i(b_{<k})$ for all t . Then the process describes a function $\tau_i = b_{<k} \uparrow 0^\infty$, where the value at the gap k is undefined. Replacing the value at k by 0 and 1 yields two functions $\tau_i^{(0)} = b0^\infty$ and $\tau_i^{(1)} = b_{k:=1}0^\infty$, which differ only at k and on which σ_i converges to the same hypothesis. Thus σ_i does not learn the class $V_i := \{\tau_i^{(0)}, \tau_i^{(1)}\}$ in the limit.

Both cases combined imply $V \notin \text{LIM}$.

A BC strategy S for $V = \bigcup_i V_i$ works as follows. Let $f \in V$. On input f^n the strategy outputs a Gödel number of the function

$$g_n(x) = \begin{cases} f(x) & \text{if } x \leq n, \\ \tau_{f(0)}(x) & \text{otherwise.} \end{cases}$$

By definition of V , f is generated by the process running for $i = f(0)$. If $f(0)$ leads to Case 1 then $f = \tau_{f(0)}$, and g_n equals f for all n . If $f(0)$ leads to Case 2 with a forever unfilled gap at k , then g_n will be equal to the correct one of $\tau_i^{(0)}$ or $\tau_i^{(1)}$ for all $n \geq k$. Intuitively, the prefix received by S eventually grows long enough to reveal the value $f(k)$. In both cases S converges to f , but it outputs a different Gödel number for every f^n because g_n contains the “hard-coded” values $f(0), \dots, f(n)$. Therefore S is a BC strategy but not a LIM strategy for V .

2.9.1 Enumerating enough total strategies

For the construction of σ we need the function $r\text{-limr}$ from the effective version of Lemma R for LIM.

definition $r\text{-sigma} \equiv Cn \ 2 \ r\text{-phi} [Cn \ 2 \ r\text{-limr} [Id \ 2 \ 0], Id \ 2 \ 1]$

lemma $r\text{-sigma-recfn}$: $recfn \ 2 \ r\text{-sigma}$

unfolding $r\text{-sigma-def}$ using $r\text{-limr-recfn}$ by $simp$

lemma *r-sigma*: $eval\ r\text{-}sigma\ [i, x] = \varphi\ (the\ (eval\ r\text{-}limr\ [i]))\ x$
unfolding *r-sigma-def phi-def* **using** *r-sigma-recfn r-limr-total r-limr-recfn*
by *simp*

lemma *r-sigma-total*: $total\ r\text{-}sigma$
using *r-sigma r-limr r-sigma-recfn totalI2[of r-sigma]* **by** *simp*

abbreviation *sigma* :: $partial2\ (\langle\sigma\rangle)$ **where**
 $\sigma\ i\ x \equiv eval\ r\text{-}sigma\ [i, x]$

lemma *sigma*: $\sigma\ i = \varphi\ (the\ (eval\ r\text{-}limr\ [i]))$
using *r-sigma* **by** *simp*

The numbering σ does indeed enumerate enough total strategies for every LIM learning problem.

lemma *learn-lim-sigma*:
assumes *learn-lim* $\psi\ U\ (\varphi\ i)$
shows *learn-lim* $\psi\ U\ (\sigma\ i)$
using *assms sigma r-limr* **by** *simp*

2.9.2 The diagonalization process

The following function represents the process described above. It computes the next state from a given state (b, k) .

definition *r-next* \equiv
 $Cn\ 1\ r\text{-}ifeq$
 $[Cn\ 1\ r\text{-}sigma\ [Cn\ 1\ r\text{-}hd\ [r\text{-}pdec1],\ r\text{-}pdec1],$
 $Cn\ 1\ r\text{-}sigma\ [Cn\ 1\ r\text{-}hd\ [r\text{-}pdec1],\ Cn\ 1\ r\text{-}take\ [r\text{-}pdec2,\ r\text{-}pdec1]],$
 $Cn\ 1\ r\text{-}ifeq$
 $[Cn\ 1\ r\text{-}sigma\ [Cn\ 1\ r\text{-}hd\ [r\text{-}pdec1],\ Cn\ 1\ r\text{-}update\ [r\text{-}pdec1,\ r\text{-}pdec2,\ r\text{-}const\ 1]],$
 $Cn\ 1\ r\text{-}sigma\ [Cn\ 1\ r\text{-}hd\ [r\text{-}pdec1],\ Cn\ 1\ r\text{-}take\ [r\text{-}pdec2,\ r\text{-}pdec1]],$
 $Cn\ 1\ r\text{-}prod\text{-}encode\ [Cn\ 1\ r\text{-}snoc\ [r\text{-}pdec1,\ Z],\ r\text{-}pdec2],$
 $Cn\ 1\ r\text{-}prod\text{-}encode$
 $[Cn\ 1\ r\text{-}snoc$
 $[Cn\ 1\ r\text{-}update\ [r\text{-}pdec1,\ r\text{-}pdec2,\ r\text{-}const\ 1],\ Z],\ Cn\ 1\ r\text{-}length\ [r\text{-}pdec1]]],$
 $Cn\ 1\ r\text{-}prod\text{-}encode\ [Cn\ 1\ r\text{-}snoc\ [r\text{-}pdec1,\ Z],\ Cn\ 1\ r\text{-}length\ [r\text{-}pdec1]]]$

lemma *r-next-recfn*: $recfn\ 1\ r\text{-}next$
unfolding *r-next-def* **using** *r-sigma-recfn* **by** *simp*

The three conditions distinguished in *r-next* correspond to Steps 1, 2, and 3 of the process: hypothesis change when the gap is filled with 0; hypothesis change when the gap is filled with 1; or no hypothesis change either way.

abbreviation *change-on-0* $b\ k \equiv \sigma\ (e\text{-}hd\ b)\ b \neq \sigma\ (e\text{-}hd\ b)\ (e\text{-}take\ k\ b)$

abbreviation *change-on-1* $b\ k \equiv$
 $\sigma\ (e\text{-}hd\ b)\ b = \sigma\ (e\text{-}hd\ b)\ (e\text{-}take\ k\ b) \wedge$
 $\sigma\ (e\text{-}hd\ b)\ (e\text{-}update\ b\ k\ 1) \neq \sigma\ (e\text{-}hd\ b)\ (e\text{-}take\ k\ b)$

abbreviation *change-on-neither* $b\ k \equiv$
 $\sigma\ (e\text{-}hd\ b)\ b = \sigma\ (e\text{-}hd\ b)\ (e\text{-}take\ k\ b) \wedge$
 $\sigma\ (e\text{-}hd\ b)\ (e\text{-}update\ b\ k\ 1) = \sigma\ (e\text{-}hd\ b)\ (e\text{-}take\ k\ b)$

lemma *change-conditions*:

obtains

(*on-0*) *change-on-0* b k
| (*on-1*) *change-on-1* b k
| (*neither*) *change-on-neither* b k
by *auto*

lemma *r-next*:

assumes $arg = prod-encode$ (b , k)

shows *change-on-0* b $k \implies eval$ *r-next* [arg] $\Downarrow = prod-encode$ ($e-snoc$ b 0 , $e-length$ b)

and *change-on-1* b $k \implies$

$eval$ *r-next* [arg] $\Downarrow = prod-encode$ ($e-update$ b k 1) 0 , $e-length$ b)

and *change-on-neither* b $k \implies eval$ *r-next* [arg] $\Downarrow = prod-encode$ ($e-snoc$ b 0 , k)

proof –

let $?bhd = Cn$ 1 *r-hd* [$r-pdec1$]

let $?bup = Cn$ 1 *r-update* [$r-pdec1$, $r-pdec2$, $r-const$ 1]

let $?bk = Cn$ 1 *r-take* [$r-pdec2$, $r-pdec1$]

let $?bap = Cn$ 1 *r-snoc* [$r-pdec1$, Z]

let $?len = Cn$ 1 *r-length* [$r-pdec1$]

let $?thenthen = Cn$ 1 *r-prod-encode* [$?bap$, $r-pdec2$]

let $?thenelse = Cn$ 1 *r-prod-encode* [Cn 1 *r-snoc* [$?bup$, Z], $?len$]

let $?else = Cn$ 1 *r-prod-encode* [$?bap$, $?len$]

have bhd : $eval$ $?bhd$ [arg] $\Downarrow = e-hd$ b

using *assms* **by** *simp*

have bup : $eval$ $?bup$ [arg] $\Downarrow = e-update$ b k 1

using *assms* **by** *simp*

have bk : $eval$ $?bk$ [arg] $\Downarrow = e-take$ k b

using *assms* **by** *simp*

have bap : $eval$ $?bap$ [arg] $\Downarrow = e-snoc$ b 0

using *assms* **by** *simp*

have len : $eval$ $?len$ [arg] $\Downarrow = e-length$ b

using *assms* **by** *simp*

have $else-$: $eval$ $?else$ [arg] $\Downarrow = prod-encode$ ($e-snoc$ b 0 , $e-length$ b)

using bap len **by** *simp*

have $thenthen$: $eval$ $?thenthen$ [arg] $\Downarrow = prod-encode$ ($e-snoc$ b 0 , k)

using bap *assms* **by** *simp*

have $thenelse$: $eval$ $?thenelse$ [arg] $\Downarrow = prod-encode$ ($e-snoc$ ($e-update$ b k 1) 0 , $e-length$ b)

using bup len **by** *simp*

have $then-$:

eval

(Cn 1 *r-ifeq* [Cn 1 *r-sigma* [$?bhd$, $?bup$], Cn 1 *r-sigma* [$?bhd$, $?bk$], $?thenthen$, $?thenelse$])

[arg] $\Downarrow =$

(*if the* (σ ($e-hd$ b) ($e-update$ b k 1)) = *the* (σ ($e-hd$ b) ($e-take$ k b))

then $prod-encode$ ($e-snoc$ b 0 , k)

else $prod-encode$ ($e-snoc$ ($e-update$ b k 1) 0 , $e-length$ b))

(**is** $eval$ $?then$ [arg] $\Downarrow = ?then-eval$)

using bhd bup bk $thenthen$ $thenelse$ *r-sigma* *r-sigma-recfn* *r-limr* *R1-imp-total1* **by** *simp*

have $*$: $eval$ *r-next* [arg] $\Downarrow =$

(*if the* (σ ($e-hd$ b) b) = *the* (σ ($e-hd$ b) ($e-take$ k b))

then $?then-eval$

else $prod-encode$ ($e-snoc$ b 0 , $e-length$ b))

unfolding *r-next-def*

using bhd bk $then-$ $else-$ *r-sigma* *r-sigma-recfn* *r-limr* *R1-imp-total1* *assms*

by *simp*

have *r-sigma-neq*: $eval$ *r-sigma* [x_1 , y_1] $\neq eval$ *r-sigma* [x_2 , y_2] \longleftrightarrow

the ($eval$ *r-sigma* [x_1 , y_1]) \neq *the* ($eval$ *r-sigma* [x_2 , y_2])

```

    for  $x_1 y_1 x_2 y_2$ 
    using  $r\text{-sigma } r\text{-limr } totalE[OF r\text{-sigma-total } r\text{-sigma-recfn}] r\text{-sigma-recfn } r\text{-sigma-total}$ 
    by ( $metis One\text{-nat-def } Suc\text{-1 } length\text{-Cons } list.size(3) option.expand$ )
  {
    assume  $change\text{-on-0 } b k$ 
    then show  $eval r\text{-next } [arg] \Downarrow = prod\text{-encode } (e\text{-snoc } b 0, e\text{-length } b)$ 
      using  $* r\text{-sigma-neq}$  by  $simp$ 
  next
    assume  $change\text{-on-1 } b k$ 
    then show  $eval r\text{-next } [arg] \Downarrow = prod\text{-encode } (e\text{-snoc } (e\text{-update } b k 1) 0, e\text{-length } b)$ 
      using  $* r\text{-sigma-neq}$  by  $simp$ 
  next
    assume  $change\text{-on-neither } b k$ 
    then show  $eval r\text{-next } [arg] \Downarrow = prod\text{-encode } (e\text{-snoc } b 0, k)$ 
      using  $* r\text{-sigma-neq}$  by  $simp$ 
  }
qed

```

```

lemma  $r\text{-next-total: total } r\text{-next}$ 
proof (rule  $totalI1$ )
  show  $recfn 1 r\text{-next}$ 
    using  $r\text{-next-recfn}$  by  $simp$ 
  show  $eval r\text{-next } [x] \Downarrow$  for  $x$ 
  proof -
    obtain  $b k$  where  $x = prod\text{-encode } (b, k)$ 
    using  $prod\text{-encode-pdec'[of } x]$  by  $metis$ 
    then show  $?thesis$  using  $r\text{-next}$  by  $fast$ 
  qed
qed

```

The next function computes the state of the process after any number of iterations.

```

definition  $r\text{-state} \equiv$ 
  Pr 1
  ( $Cn 1 r\text{-prod-encode } [Cn 1 r\text{-snoc } [Cn 1 r\text{-singleton-encode } [Id 1 0], Z], r\text{-const } 1]$ )
  ( $Cn 3 r\text{-next } [Id 3 1]$ )

```

```

lemma  $r\text{-state-recfn: recfn } 2 r\text{-state}$ 
  unfolding  $r\text{-state-def}$  using  $r\text{-next-recfn}$  by  $simp$ 

```

```

lemma  $r\text{-state-at-0: eval } r\text{-state } [0, i] \Downarrow = prod\text{-encode } (list\text{-encode } [i, 0], 1)$ 
proof -
  let  $?f = Cn 1 r\text{-prod-encode } [Cn 1 r\text{-snoc } [Cn 1 r\text{-singleton-encode } [Id 1 0], Z], r\text{-const } 1]$ 
  have  $eval r\text{-state } [0, i] = eval ?f [i]$ 
    unfolding  $r\text{-state-def}$  using  $r\text{-next-recfn}$  by  $simp$ 
  also have  $\dots \Downarrow = prod\text{-encode } (list\text{-encode } [i, 0], 1)$ 
    by ( $simp add: list\text{-decode-singleton}$ )
  finally show  $?thesis$  .
qed

```

```

lemma  $r\text{-state-total: total } r\text{-state}$ 
  unfolding  $r\text{-state-def}$ 
  using  $r\text{-next-recfn } totalE[OF r\text{-next-total } r\text{-next-recfn}] totalI3[of } Cn 3 r\text{-next } [Id 3 1]]$ 
  by ( $intro Pr\text{-total}$ )  $auto$ 

```

We call the components of a state (b, k) the *block* b and the *gap* k .

```

definition  $block :: nat \Rightarrow nat \Rightarrow nat$  where

```

$block\ i\ t \equiv pdec1\ (the\ (eval\ r-state\ [t,\ i]))$

definition $gap :: nat \Rightarrow nat \Rightarrow nat$ **where**
 $gap\ i\ t \equiv pdec2\ (the\ (eval\ r-state\ [t,\ i]))$

lemma *state-at-0*:

$block\ i\ 0 = list-encode\ [i,\ 0]$

$gap\ i\ 0 = 1$

unfolding *block-def gap-def r-state-at-0* **by** *simp-all*

Some lemmas describing the behavior of blocks and gaps in one iteration of the process:

lemma *state-Suc*:

assumes $b = block\ i\ t$ **and** $k = gap\ i\ t$

shows $block\ i\ (Suc\ t) = pdec1\ (the\ (eval\ r-next\ [prod-encode\ (b,\ k)]))$

and $gap\ i\ (Suc\ t) = pdec2\ (the\ (eval\ r-next\ [prod-encode\ (b,\ k)]))$

proof –

have $eval\ r-state\ [Suc\ t,\ i] =$

$eval\ (Cn\ 3\ r-next\ [Id\ 3\ 1])\ [t,\ the\ (eval\ r-state\ [t,\ i]),\ i]$

using *r-state-recfn r-next-recfn totalE[OF r-state-total r-state-recfn, of [t, i]]*

by (*simp add: r-state-def*)

also have $\dots = eval\ r-next\ [the\ (eval\ r-state\ [t,\ i])]$

using *r-next-recfn* **by** *simp*

also have $\dots = eval\ r-next\ [prod-encode\ (b,\ k)]$

using *assms block-def gap-def* **by** *simp*

finally have $eval\ r-state\ [Suc\ t,\ i] = eval\ r-next\ [prod-encode\ (b,\ k)]$.

then show

$block\ i\ (Suc\ t) = pdec1\ (the\ (eval\ r-next\ [prod-encode\ (b,\ k)]))$

$gap\ i\ (Suc\ t) = pdec2\ (the\ (eval\ r-next\ [prod-encode\ (b,\ k)]))$

by (*simp add: block-def, simp add: gap-def*)

qed

lemma *gap-Suc*:

assumes $b = block\ i\ t$ **and** $k = gap\ i\ t$

shows $change-on-0\ b\ k \Longrightarrow gap\ i\ (Suc\ t) = e-length\ b$

and $change-on-1\ b\ k \Longrightarrow gap\ i\ (Suc\ t) = e-length\ b$

and $change-on-neither\ b\ k \Longrightarrow gap\ i\ (Suc\ t) = k$

using *assms r-next state-Suc* **by** *simp-all*

lemma *block-Suc*:

assumes $b = block\ i\ t$ **and** $k = gap\ i\ t$

shows $change-on-0\ b\ k \Longrightarrow block\ i\ (Suc\ t) = e-snoc\ b\ 0$

and $change-on-1\ b\ k \Longrightarrow block\ i\ (Suc\ t) = e-snoc\ (e-update\ b\ k\ 1)\ 0$

and $change-on-neither\ b\ k \Longrightarrow block\ i\ (Suc\ t) = e-snoc\ b\ 0$

using *assms r-next state-Suc* **by** *simp-all*

Non-gap positions in the block remain unchanged after an iteration.

lemma *block-stable*:

assumes $j < e-length\ (block\ i\ t)$ **and** $j \neq gap\ i\ t$

shows $e-nth\ (block\ i\ t)\ j = e-nth\ (block\ i\ (Suc\ t))\ j$

proof –

from *change-conditions[of block i t gap i t]* **show** *?thesis*

using *assms block-Suc gap-Suc*

by (*cases, (simp-all add: nth-append)*)

qed

Next are some properties of *block* and *gap*.

```

lemma gap-in-block: gap i t < e-length (block i t)
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
  case (Suc t)
  with change-conditions[of block i t gap i t] show ?case
  proof (cases)
    case on-0
    then show ?thesis by (simp add: block-Suc(1) gap-Suc(1))
  next
    case on-1
    then show ?thesis by (simp add: block-Suc(2) gap-Suc(2))
  next
    case neither
    then show ?thesis using Suc.IH block-Suc(3) gap-Suc(3) by force
  qed
qed

lemma length-block: e-length (block i t) = Suc (Suc t)
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
  case (Suc t)
  with change-conditions[of block i t gap i t] show ?case
  by (cases, simp-all add: block-Suc gap-Suc)
qed

lemma gap-gr0: gap i t > 0
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
  case (Suc t)
  with change-conditions[of block i t gap i t] show ?case
  using length-block by (cases, simp-all add: block-Suc gap-Suc)
qed

lemma hd-block: e-hd (block i t) = i
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
  case (Suc t)
  from change-conditions[of block i t gap i t] show ?case
  proof (cases)
    case on-0
    then show ?thesis
    using Suc block-Suc(1) length-block by (metis e-hd-snoc gap-Suc(1) gap-gr0)
  next
    case on-1
    let ?b = block i t and ?k = gap i t
    have ?k > 0
    using gap-gr0 Suc by simp
    then have e-nth (e-update ?b ?k 1) 0 = e-nth ?b 0

```

```

    by simp
  then have *: e-hd (e-update ?b ?k 1) = e-hd ?b
    using e-hd-nth0 gap-Suc(2)[of - i t] gap-gr0 on-1 by (metis e-length-update)
  from on-1 have block i (Suc t) = e-snoc (e-update ?b ?k 1) 0
    by (simp add: block-Suc(2))
  then show ?thesis
    using e-hd-0 e-hd-snoc Suc length-block ⟨?k > 0⟩ *
    by (metis e-length-update gap-Suc(2) gap-gr0 on-1)
next
case neither
then show ?thesis
  by (metis Suc block-stable e-hd-nth0 gap-gr0 length-block not-gr0 zero-less-Suc)
qed
qed

```

Formally, a block always ends in zero, even if it ends in a gap.

```

lemma last-block: e-nth (block i t) (gap i t) = 0
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
case (Suc t)
from change-conditions[of block i t gap i t] show ?case
proof cases
  case on-0
  then show ?thesis using Suc by (simp add: block-Suc(1) gap-Suc(1))
next
  case on-1
  then show ?thesis using Suc by (simp add: block-Suc(2) gap-Suc(2) nth-append)
next
  case neither
  then have
    block i (Suc t) = e-snoc (block i t) 0
    gap i (Suc t) = gap i t
    by (simp-all add: gap-Suc(3) block-Suc(3))
  then show ?thesis
    using Suc gap-in-block by (simp add: nth-append)
qed
qed

```

```

lemma gap-le-Suc: gap i t ≤ gap i (Suc t)
  using change-conditions[of block i t gap i t]
  gap-Suc gap-in-block less-imp-le[of gap i t e-length (block i t)]
  by (cases) simp-all

```

```

lemma gap-monotone:
  assumes  $t_1 \leq t_2$ 
  shows  $gap\ i\ t_1 \leq gap\ i\ t_2$ 
proof -
  have  $gap\ i\ t_1 \leq gap\ i\ (t_1 + j)$  for  $j$ 
  proof (induction j)
    case 0
    then show ?case by simp
  next
  case (Suc j)
  then show ?case using gap-le-Suc dual-order.trans by fastforce

```

```

qed
then show ?thesis using assms le-Suc-ex by blast
qed

```

We need some lemmas relating the shape of the next state to the hypothesis change conditions in Steps 1, 2, and 3.

```

lemma state-change-on-neither:
  assumes gap i (Suc t) = gap i t
  shows change-on-neither (block i t) (gap i t)
  and block i (Suc t) = e-snoc (block i t) 0
proof -
  let ?b = block i t and ?k = gap i t
  have ?k < e-length ?b
  using gap-in-block by simp
  from change-conditions[of ?b ?k] show change-on-neither (block i t) (gap i t)
  proof (cases)
    case on-0
    then show ?thesis
    using ‹?k < e-length ?b› assms gap-Suc(1) by auto
  next
    case on-1
    then show ?thesis using assms gap-Suc(2) by (auto simp: list-update-beyond)
  next
    case neither
    then show ?thesis by simp
  qed
  then show block i (Suc t) = e-snoc (block i t) 0
  using block-Suc(3) by simp
qed

```

```

lemma state-change-on-either:
  assumes gap i (Suc t) ≠ gap i t
  shows ¬ change-on-neither (block i t) (gap i t)
  and gap i (Suc t) = e-length (block i t)
proof -
  let ?b = block i t and ?k = gap i t
  show ¬ change-on-neither (block i t) (gap i t)
  proof
    assume change-on-neither (block i t) (gap i t)
    then have gap i (Suc t) = ?k
    by (simp add: gap-Suc(3))
    with assms show False by simp
  qed
  then show gap i (Suc t) = e-length (block i t)
  using gap-Suc(1) gap-Suc(2) by blast
qed

```

Next up is the definition of τ . In every iteration the process determines $\tau_i(x)$ for some x either by appending 0 to the current block b , or by filling the current gap k . In the former case, the value is determined for $x = |b|$, in the latter for $x = k$.

For i and x the function $r\text{-detime}$ computes in which iteration the process for i determines the value $\tau_i(x)$. This is the first iteration in which the block is long enough to contain position x and in which x is not the gap. If $\tau_i(x)$ is never determined, because Case 2 is reached with $k = x$, then $r\text{-detime}$ diverges.

abbreviation *determined* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
determined $i\ x \equiv \exists t. x < e\text{-length}\ (\text{block}\ i\ t) \wedge x \neq \text{gap}\ i\ t$

lemma *determined-0*: *determined* $i\ 0$
using *gap-gr0*[*of* $i\ 0$] *gap-in-block*[*of* $i\ 0$] **by** *force*

definition *r-detime* \equiv
Mn 2
(*Cn* 3 *r-and*
[*Cn* 3 *r-less*
[*Id* 3 2, *Cn* 3 *r-length* [*Cn* 3 *r-pdec1* [*Cn* 3 *r-state* [*Id* 3 0, *Id* 3 1]]]],
Cn 3 *r-neq*
[*Id* 3 2, *Cn* 3 *r-pdec2* [*Cn* 3 *r-state* [*Id* 3 0, *Id* 3 1]]]])

lemma *r-detime-recfn*: *recfn* 2 *r-detime*
unfolding *r-detime-def* **using** *r-state-recfn* **by** *simp*

abbreviation *detime* :: *partial2* **where**
detime $i\ x \equiv \text{eval}\ r\text{-detime}\ [i, x]$

lemma *r-detime*:
shows *determined* $i\ x \implies \text{detime}\ i\ x \downarrow = (\text{LEAST}\ t. x < e\text{-length}\ (\text{block}\ i\ t) \wedge x \neq \text{gap}\ i\ t)$
and $\neg \text{determined}\ i\ x \implies \text{detime}\ i\ x \uparrow$

proof –

define *f* **where** $f =$
(*Cn* 3 *r-and*
[*Cn* 3 *r-less*
[*Id* 3 2, *Cn* 3 *r-length* [*Cn* 3 *r-pdec1* [*Cn* 3 *r-state* [*Id* 3 0, *Id* 3 1]]]],
Cn 3 *r-neq*
[*Id* 3 2, *Cn* 3 *r-pdec2* [*Cn* 3 *r-state* [*Id* 3 0, *Id* 3 1]]]])

then have $r\text{-detime} = \text{Mn}\ 2\ f$
unfolding *f-def* *r-detime-def* **by** *simp*
have *recfn* 3 *f*
unfolding *f-def* **using** *r-state-recfn* **by** *simp*

then have *total* *f*
unfolding *f-def* **using** *Cn-total* *r-state-total* *Mn-free-imp-total* **by** *simp*

have *f*: $\text{eval}\ f\ [t, i, x] \downarrow = (\text{if}\ x < e\text{-length}\ (\text{block}\ i\ t) \wedge x \neq \text{gap}\ i\ t\ \text{then}\ 0\ \text{else}\ 1)$ **for** t

proof –

let $?b = \text{Cn}\ 3\ r\text{-pdec1}\ [\text{Cn}\ 3\ r\text{-state}\ [\text{Id}\ 3\ 0, \text{Id}\ 3\ 1]]$
let $?k = \text{Cn}\ 3\ r\text{-pdec2}\ [\text{Cn}\ 3\ r\text{-state}\ [\text{Id}\ 3\ 0, \text{Id}\ 3\ 1]]$
have $\text{eval}\ ?b\ [t, i, x] \downarrow = \text{pdec1}\ (\text{the}\ (\text{eval}\ r\text{-state}\ [t, i]))$
using *r-state-recfn* *r-state-total* **by** *simp*
then have $b: \text{eval}\ ?b\ [t, i, x] \downarrow = \text{block}\ i\ t$
using *block-def* **by** *simp*
have $\text{eval}\ ?k\ [t, i, x] \downarrow = \text{pdec2}\ (\text{the}\ (\text{eval}\ r\text{-state}\ [t, i]))$
using *r-state-recfn* *r-state-total* **by** *simp*
then have $k: \text{eval}\ ?k\ [t, i, x] \downarrow = \text{gap}\ i\ t$
using *gap-def* **by** *simp*
have *eval*
(*Cn* 3 *r-neq* [*Id* 3 2, *Cn* 3 *r-pdec2* [*Cn* 3 *r-state* [*Id* 3 0, *Id* 3 1]]])
[t, i, x] $\downarrow =$
(*if* $x \neq \text{gap}\ i\ t$ *then* 0 *else* 1)
using $b\ k$ *r-state-recfn* *r-state-total* **by** *simp*
moreover have *eval*
(*Cn* 3 *r-less*
[*Id* 3 2, *Cn* 3 *r-length* [*Cn* 3 *r-pdec1* [*Cn* 3 *r-state* [*Id* 3 0, *Id* 3 1]]]])

```

    [t, i, x] ↓=
    (if x < e-length (block i t) then 0 else 1)
    using b k r-state-recfn r-state-total by simp
    ultimately show ?thesis
    unfolding f-def using b k r-state-recfn r-state-total by simp
qed
{
  assume determined i x
  with f have ∃ t. eval f [t, i, x] ↓= 0 by simp
  then have dettime i x ↓= (LEAST t. eval f [t, i, x] ↓= 0)
    using ⟨total f⟩ ⟨r-dettime = Mn 2 f⟩ r-dettime-recfn ⟨recfn 3 f⟩
    eval-Mn-total[of 2 f [i, x]]
    by simp
  then show dettime i x ↓= (LEAST t. x < e-length (block i t) ∧ x ≠ gap i t)
    using f by simp
next
  assume ¬ determined i x
  with f have ¬ (∃ t. eval f [t, i, x] ↓= 0) by simp
  then have dettime i x ↑
    using ⟨total f⟩ ⟨r-dettime = Mn 2 f⟩ r-dettime-recfn ⟨recfn 3 f⟩
    eval-Mn-total[of 2 f [i, x]]
    by simp
  with f show dettime i x ↑ by simp
}
qed

```

lemma *r-dettimeI*:

```

  assumes x < e-length (block i t) ∧ x ≠ gap i t
  and ∧ T. x < e-length (block i T) ∧ x ≠ gap i T ⇒ t ≤ T
  shows dettime i x ↓= t

```

proof –

```

  let ?P = λ T. x < e-length (block i T) ∧ x ≠ gap i T
  have determined i x
    using assms(1) by auto
  moreover have Least ?P = t
    using assms Least-equality[of ?P t] by simp
  ultimately show ?thesis using r-dettime by simp

```

qed

lemma *r-dettime-0*: $\text{dettime } i \ 0 \downarrow= 0$

```

  using r-dettimeI[of - i 0] determined-0 gap-gr0[of i 0] gap-in-block[of i 0]
  by fastforce

```

Computing the value of $\tau_i(x)$ works by running the process *r-state* for *dettime i x* iterations and taking the value at index *x* of the resulting block.

definition *r-tau* $\equiv Cn \ 2 \ r\text{-nth} \ [Cn \ 2 \ r\text{-pdec1} \ [Cn \ 2 \ r\text{-state} \ [r\text{-dettime}, \ Id \ 2 \ 0]], \ Id \ 2 \ 1]$

lemma *r-tau-recfn*: $\text{recfn } 2 \ r\text{-tau}$

```

  unfolding r-tau-def using r-dettime-recfn r-state-recfn by simp

```

abbreviation *tau* :: *partial2* ($\langle \tau \rangle$) **where**

```

  τ i x ≡ eval r-tau [i, x]

```

lemma *tau-in-P2*: $\tau \in \mathcal{P}^2$

```

  using r-tau-recfn by auto

```

lemma tau-diverg:

assumes \neg *determined* *i x*

shows τ *i x* \uparrow

unfolding *r-tau-def* **using** *assms r-detime r-detime-recfn r-state-recfn* **by** *simp*

lemma tau-converg:

assumes *determined* *i x*

shows τ *i x* $\downarrow = e\text{-nth}$ (*block* *i* (*the* (*detime* *i x*))) *x*

proof –

from *assms* **obtain** *t* **where** *t*: *detime* *i x* $\downarrow = t$

using *r-detime(1)* **by** *blast*

then have *eval* (*Cn* 2 *r-state* [*r-detime*, *Id* 2 0]) [*i*, *x*] = *eval* *r-state* [*t*, *i*]

using *r-state-recfn r-detime-recfn* **by** *simp*

moreover have *eval* *r-state* [*t*, *i*] \downarrow

using *r-state-total r-state-recfn* **by** *simp*

ultimately have *eval* (*Cn* 2 *r-pdec1* [*Cn* 2 *r-state* [*r-detime*, *Id* 2 0]]) [*i*, *x*] =
eval *r-pdec1* [*the* (*eval* *r-state* [*t*, *i*])]

using *r-state-recfn r-detime-recfn* **by** *simp*

then show *?thesis*

unfolding *r-tau-def* **using** *r-state-recfn r-detime-recfn t block-def* **by** *simp*

qed

lemma tau-converg':

assumes *detime* *i x* $\downarrow = t$

shows τ *i x* $\downarrow = e\text{-nth}$ (*block* *i t*) *x*

using *assms tau-converg[of x i] r-detime(2)[of x i]* **by** *fastforce*

lemma tau-at-0: τ *i 0* $\downarrow = i$

proof –

have τ *i 0* $\downarrow = e\text{-nth}$ (*block* *i 0*) 0

using *tau-converg'[OF r-detime-0]* **by** *simp*

then show *?thesis* **using** *block-def* **by** (*simp add: r-state-at-0*)

qed

lemma state-unchanged:

assumes *gap* *i t* $- 1 \leq y$ **and** $y \leq t$

shows *gap* *i t* = *gap* *i y*

proof –

have *gap* *i t* = *gap* *i* (*gap* *i t* $- 1$)

proof (*induction* *t*)

case 0

then show *?case* **by** (*simp add: gap-def r-state-at-0*)

next

case (*Suc* *t*)

show *?case*

proof (*cases* *gap* *i* (*Suc* *t*) = *t* + 2)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

then show *?thesis*

using *Suc state-change-on-either(2) length-block* **by** *force*

qed

qed

moreover have *gap* *i* (*gap* *i t* $- 1$) \leq *gap* *i y*

using *assms(1) gap-monotone* **by** *simp*

```

moreover have  $gap\ i\ y \leq gap\ i\ t$ 
  using  $assms(2)$   $gap\ monotone$  by  $simp$ 
ultimately show  $?thesis$  by  $simp$ 
qed

```

The values of the non-gap indices x of every block created in the diagonalization process equal $\tau_i(x)$.

lemma $tau\ eq\ state$:

```

assumes  $j < e\ length\ (block\ i\ t)$  and  $j \neq gap\ i\ t$ 
shows  $\tau\ i\ j \downarrow = e\ nth\ (block\ i\ t)\ j$ 
using  $assms$ 
proof ( $induction\ t$ )
case  $0$ 
  then have  $j = 0$ 
    using  $gap\ gr0[of\ i\ 0]$   $gap\ in\ block[of\ i\ 0]$   $length\ block[of\ i\ 0]$  by  $simp$ 
  then have  $\tau\ (e\ hd\ (block\ i\ t))\ j \downarrow = e\ nth\ (block\ i\ (the\ (detime\ i\ 0)))\ 0$ 
    using  $determined\ 0\ tau\ converg\ hd\ block$  by  $simp$ 
  then have  $\tau\ (e\ hd\ (block\ i\ t))\ j \downarrow = e\ nth\ (block\ i\ 0)\ 0$ 
    using  $r\ dettime\ 0$  by  $simp$ 
  then show  $?case$  using  $\langle j = 0 \rangle\ r\ dettime\ 0\ tau\ converg'$  by  $simp$ 
next
case ( $Suc\ t$ )
  let  $?b = block\ i\ t$ 
  let  $?bb = block\ i\ (Suc\ t)$ 
  let  $?k = gap\ i\ t$ 
  let  $?kk = gap\ i\ (Suc\ t)$ 
  show  $?case$ 
proof ( $cases\ ?kk = ?k$ )
  case  $kk\ eq\ k$ :  $True$ 
    then have  $bb\ b0$ :  $?bb = e\ snoc\ ?b\ 0$ 
      using  $state\ change\ on\ neither$  by  $simp$ 
    show  $\tau\ i\ j \downarrow = e\ nth\ ?bb\ j$ 
proof ( $cases\ j < e\ length\ ?b$ )
  case  $True$ 
    then have  $e\ nth\ ?bb\ j = e\ nth\ ?b\ j$ 
      using  $bb\ b0$  by ( $simp\ add$ :  $nth\ append$ )
    moreover have  $j \neq ?k$ 
      using  $Suc\ kk\ eq\ k$  by  $simp$ 
    ultimately show  $?thesis$  using  $Suc\ True$  by  $simp$ 
  next
  case  $False$ 
    then have  $j$ :  $j = e\ length\ ?b$ 
      using  $Suc.prem1s(1)$   $length\ block$  by  $auto$ 
    then have  $e\ nth\ ?bb\ j = 0$ 
      using  $bb\ b0$  by  $simp$ 
    have  $detime\ i\ j \downarrow = Suc\ t$ 
proof ( $rule\ r\ dettimeI$ )
      show  $j < e\ length\ ?bb \wedge j \neq ?kk$ 
        using  $Suc.prem1s(1,2)$  by  $linarith$ 
      show  $\bigwedge T. j < e\ length\ (block\ i\ T) \wedge j \neq gap\ i\ T \implies Suc\ t \leq T$ 
        using  $length\ block\ j$  by  $simp$ 
    qed
    with  $tau\ converg'$  show  $?thesis$  by  $simp$ 
qed
next
case  $False$ 

```

```

then have kk-lenb:  $?kk = e\text{-length } ?b$ 
  using state-change-on-either by simp
then show ?thesis
proof (cases  $j = ?k$ )
  case j-eq-k: True
  have detime  $i j \downarrow = \text{Suc } t$ 
  proof (rule r-detimeI)
    show  $j < e\text{-length } ?bb \wedge j \neq ?kk$ 
      using Suc.prems(1,2) by simp
    show  $\text{Suc } t \leq T$  if  $j < e\text{-length } (\text{block } i T) \wedge j \neq \text{gap } i T$  for  $T$ 
    proof (rule ccontr)
      assume  $\neg (\text{Suc } t \leq T)$ 
      then have  $T < \text{Suc } t$  by simp
      then show False
    proof (cases  $T < ?k - 1$ )
      case True
      then have  $e\text{-length } (\text{block } i T) = T + 2$ 
        using length-block by simp
      then have  $e\text{-length } (\text{block } i T) < ?k + 1$ 
        using True by simp
      then have  $e\text{-length } (\text{block } i T) \leq ?k$  by simp
      then have  $e\text{-length } (\text{block } i T) \leq j$ 
        using j-eq-k by simp
      then show False
        using that by simp
      next
      case False
      then have  $?k - 1 \leq T$  and  $T \leq t$ 
        using  $\langle T < \text{Suc } t \rangle$  by simp-all
      with state-unchanged have  $\text{gap } i t = \text{gap } i T$  by blast
      then show False
        using j-eq-k that by simp
    qed
  qed
  qed
  then show ?thesis using tau-converg' by simp
next
  case False
  then have  $j < e\text{-length } ?b$ 
    using kk-lenb Suc.prems(1,2) length-block by auto
  then show ?thesis using Suc False block-stable by fastforce
  qed
  qed
  qed

```

lemma *tau-eq-state'*:

```

assumes  $j < t + 2$  and  $j \neq \text{gap } i t$ 
shows  $\tau i j \downarrow = e\text{-nth } (\text{block } i t) j$ 
using assms tau-eq-state length-block by simp

```

We now consider the two cases described in the proof sketch. In Case 2 there is a gap that never gets filled, or equivalently there is a rightmost gap.

abbreviation *case-two* $i \equiv (\exists t. \forall T. \text{gap } i T \leq \text{gap } i t)$

abbreviation *case-one* $i \equiv \neg \text{case-two } i$

Another characterization of Case 2 is that from some iteration on only *change-on-neither* holds.

lemma *case-two-iff-forever-neither*:

case-two $i \longleftrightarrow (\exists t. \forall T \geq t. \text{change-on-neither } (\text{block } i \ T) \ (\text{gap } i \ T))$

proof

assume $\exists t. \forall T \geq t. \text{change-on-neither } (\text{block } i \ T) \ (\text{gap } i \ T)$

then obtain t **where** $t: \forall T \geq t. \text{change-on-neither } (\text{block } i \ T) \ (\text{gap } i \ T)$

by *auto*

have $(\text{gap } i \ T) \leq (\text{gap } i \ t)$ **for** T

proof (*cases* $T \leq t$)

case *True*

then show *?thesis* **using** *gap-monotone* **by** *simp*

next

case *False*

then show *?thesis*

proof (*induction* T)

case 0

then show *?case* **by** *simp*

next

case (*Suc* T)

with t **have** *change-on-neither* $((\text{block } i \ T)) \ ((\text{gap } i \ T))$

by *simp*

then show *?case*

using *Suc.IH* *state-change-on-either(1)*[*of* $i \ T$] *gap-monotone*[*of* $T \ t \ i$]

by *metis*

qed

qed

then show $\exists t. \forall T. \text{gap } i \ T \leq \text{gap } i \ t$

by *auto*

next

assume $\exists t. \forall T. \text{gap } i \ T \leq \text{gap } i \ t$

then obtain t **where** $t: \forall T. \text{gap } i \ T \leq \text{gap } i \ t$

by *auto*

have *change-on-neither* $(\text{block } i \ T) \ (\text{gap } i \ T)$ **if** $T \geq t$ **for** T

proof –

have $T: (\text{gap } i \ T) \geq (\text{gap } i \ t)$

using *gap-monotone* **that** **by** *simp*

show *?thesis*

proof (*rule* *ccontr*)

assume $\neg \text{change-on-neither } (\text{block } i \ T) \ (\text{gap } i \ T)$

then have *change-on-0* $(\text{block } i \ T) \ (\text{gap } i \ T) \vee \text{change-on-1}$ $(\text{block } i \ T) \ (\text{gap } i \ T)$

by *simp*

then have $\text{gap } i \ (\text{Suc } T) > \text{gap } i \ T$

using *gap-le-Suc*[*of* i] *state-change-on-either(2)*[*of* i] *state-change-on-neither(1)*[*of* i]
dual-order.strict-iff-order

by *blast*

with T **have** $\text{gap } i \ (\text{Suc } T) > \text{gap } i \ t$ **by** *simp*

with t **show** *False*

using *not-le* **by** *auto*

qed

qed

then show $\exists t. \forall T \geq t. \text{change-on-neither } (\text{block } i \ T) \ (\text{gap } i \ T)$

by *auto*

qed

In Case 1, τ_i is total.

lemma *case-one-tau-total*:
assumes *case-one i*
shows $\tau i x \downarrow$
proof (*cases x = gap i x*)
case *True*
from *assms* **have** $\forall t. \exists T. \text{gap } i \ T > \text{gap } i \ t$
using *le-less-linear gap-def[of i x]* **by** *blast*
then obtain *T* **where** $T: \text{gap } i \ T > \text{gap } i \ x$
by *auto*
then have $T > x$
using *gap-monotone leD le-less-linear* **by** *blast*
then have $x < T + 2$ **by** *simp*
moreover from *T True* **have** $x \neq \text{gap } i \ T$ **by** *simp*
ultimately show *?thesis* **using** *tau-eq-state'* **by** *simp*
next
case *False*
moreover have $x < x + 2$ **by** *simp*
ultimately show *?thesis* **using** *tau-eq-state'* **by** *blast*
qed

In Case 2, τ_i is undefined only at the gap that never gets filled.

lemma *case-two-tau-not-quite-total*:
assumes $\forall T. \text{gap } i \ T \leq \text{gap } i \ t$
shows $\tau i (\text{gap } i \ t) \uparrow$
and $x \neq \text{gap } i \ t \implies \tau i x \downarrow$
proof –
let $?k = \text{gap } i \ t$
have $\neg \text{determined } i \ ?k$
proof
assume *determined i ?k*
then obtain *T* **where** $T: ?k < e\text{-length } (\text{block } i \ T) \wedge ?k \neq \text{gap } i \ T$
by *auto*
with *assms* **have** *snd-le: gap i T < ?k*
by (*simp add: dual-order.strict-iff-order*)
then have $T < t$
using *gap-monotone* **by** (*metis leD le-less-linear*)
from *T length-block* **have** $?k < T + 2$ **by** *simp*
moreover have $?k \neq T + 1$
using *T state-change-on-either(2) <T < t> state-unchanged*
by (*metis Suc-eq-plus1 Suc-leI add-diff-cancel-right' le-add1 nat-neq-iff*)
ultimately have $?k \leq T$ **by** *simp*
then have $\text{gap } i \ T = \text{gap } i \ ?k$
using *state-unchanged[of i T ?k] <?k < T + 2> snd-le* **by** *simp*
then show *False*
by (*metis diff-le-self state-unchanged leD nat-le-linear gap-monotone snd-le*)
qed
with *tau-diverg* **show** $\tau i ?k \uparrow$ **by** *simp*

assume $x \neq ?k$
show $\tau i x \downarrow$
proof (*cases x < t + 2*)
case *True*
with $\langle x \neq ?k \rangle$ *tau-eq-state'* **show** *?thesis* **by** *simp*
next
case *False*
then have $\text{gap } i \ x = ?k$

using *assms* **by** (*simp add: dual-order.antisym gap-monotone*)
with $\langle x \neq ?k \rangle$ **have** $x \neq \text{gap } i \ x$ **by** *simp*
then show *?thesis* **using** *tau-eq-state*[*of x x*] **by** *simp*
qed
qed

lemma *case-two-tau-almost-total*:
assumes $\exists t. \forall T. \text{gap } i \ T \leq \text{gap } i \ t$ (**is** $\exists t. ?P \ t$)
shows $\tau \ i \ (\text{gap } i \ (\text{Least } ?P)) \uparrow$
and $x \neq \text{gap } i \ (\text{Least } ?P) \implies \tau \ i \ x \downarrow$
proof –
from *assms* **have** $?P \ (\text{Least } ?P)$
using *LeastI-ex*[*of ?P*] **by** *simp*
then show $\tau \ i \ (\text{gap } i \ (\text{Least } ?P)) \uparrow$ **and** $x \neq \text{gap } i \ (\text{Least } ?P) \implies \tau \ i \ x \downarrow$
using *case-two-tau-not-quite-total* **by** *simp-all*
qed

Some more properties of τ .

lemma *init-tau-gap*: $(\tau \ i) \triangleright (\text{gap } i \ t - 1) = e\text{-take } (\text{gap } i \ t) \ (\text{block } i \ t)$

proof (*intro initI'*)
show *1*: $e\text{-length } (e\text{-take } (\text{gap } i \ t) \ (\text{block } i \ t)) = \text{Suc } (\text{gap } i \ t - 1)$
proof –
have $\text{gap } i \ t > 0$
using *gap-gr0* **by** *simp*
moreover have $\text{gap } i \ t < e\text{-length } (\text{block } i \ t)$
using *gap-in-block* **by** *simp*
ultimately have $e\text{-length } (e\text{-take } (\text{gap } i \ t) \ (\text{block } i \ t)) = \text{gap } i \ t$
by *simp*
then show *?thesis* **using** *gap-gr0* **by** *simp*
qed

show $\tau \ i \ x \downarrow = e\text{-nth } (e\text{-take } (\text{gap } i \ t) \ (\text{block } i \ t)) \ x$ **if** $x < \text{Suc } (\text{gap } i \ t - 1)$ **for** x

proof –
have *x-le*: $x < \text{gap } i \ t$
using *that gap-gr0* **by** *simp*
then have $x < e\text{-length } (\text{block } i \ t)$
using *gap-in-block less-trans* **by** *blast*
then have $\tau \ i \ x \downarrow = e\text{-nth } (\text{block } i \ t) \ x$
using *x-le tau-eq-state* **by** *auto*
have $x < e\text{-length } (e\text{-take } (\text{gap } i \ t) \ (\text{block } i \ t))$
using *x-le 1* **by** *simp*
then have $e\text{-nth } (\text{block } i \ t) \ x = e\text{-nth } (e\text{-take } (\text{gap } i \ t) \ (\text{block } i \ t)) \ x$
using *x-le* **by** *simp*
then show *?thesis* **using** $*$ **by** *simp*
qed
qed

lemma *change-on-0-init-tau*:

assumes *change-on-0* $(\text{block } i \ t) \ (\text{gap } i \ t)$
shows $(\tau \ i) \triangleright (t + 1) = \text{block } i \ t$
proof (*intro initI'*)
let $?b = \text{block } i \ t$ **and** $?k = \text{gap } i \ t$
show $e\text{-length } (\text{block } i \ t) = \text{Suc } (t + 1)$
using *length-block* **by** *simp*
show $(\tau \ i) \ x \downarrow = e\text{-nth } (\text{block } i \ t) \ x$ **if** $x < \text{Suc } (t + 1)$ **for** x
proof (*cases x = ?k*)
case *True*

have $gap\ i\ (Suc\ t) = e\text{-length}\ ?b$ **and** $b: block\ i\ (Suc\ t) = e\text{-snoc}\ ?b\ 0$
using $gap\text{-}Suc(1)$ $block\text{-}Suc(1)$ $assms$ **by** $simp\text{-}all$
then have $x < e\text{-length}\ (block\ i\ (Suc\ t))\ x \neq gap\ i\ (Suc\ t)$
using $that\ length\text{-}block$ **by** $simp\text{-}all$
then have $\tau\ i\ x \downarrow = e\text{-nth}\ (block\ i\ (Suc\ t))\ x$
using $tau\text{-}eq\text{-}state$ **by** $simp$
then show $?thesis$ **using** $that\ assms\ b$ **by** $(simp\ add: nth\text{-}append)$
next
case $False$
then show $?thesis$ **using** $that\ assms\ tau\text{-}eq\text{-}state'$ **by** $simp$
qed
qed

lemma $change\text{-}on\text{-}0\text{-}hyp\text{-}change:$

assumes $change\text{-}on\text{-}0\ (block\ i\ t)\ (gap\ i\ t)$
shows $\sigma\ i\ ((\tau\ i) \triangleright (t + 1)) \neq \sigma\ i\ ((\tau\ i) \triangleright (gap\ i\ t - 1))$
using $assms\ hd\text{-}block\ init\text{-}tau\text{-}gap\ change\text{-}on\text{-}0\text{-}init\text{-}tau$ **by** $simp$

lemma $change\text{-}on\text{-}1\text{-}init\text{-}tau:$

assumes $change\text{-}on\text{-}1\ (block\ i\ t)\ (gap\ i\ t)$
shows $(\tau\ i) \triangleright (t + 1) = e\text{-update}\ (block\ i\ t)\ (gap\ i\ t)\ 1$

proof $(intro\ initI')$

let $?b = block\ i\ t$ **and** $?k = gap\ i\ t$
show $e\text{-length}\ (e\text{-update}\ ?b\ ?k\ 1) = Suc\ (t + 1)$
using $length\text{-}block$ **by** $simp$
show $(\tau\ i)\ x \downarrow = e\text{-nth}\ (e\text{-update}\ ?b\ ?k\ 1)\ x$ **if** $x < Suc\ (t + 1)$ **for** x
proof $(cases\ x = ?k)$

case $True$

have $gap\ i\ (Suc\ t) = e\text{-length}\ ?b$ **and** $b: block\ i\ (Suc\ t) = e\text{-snoc}\ (e\text{-update}\ ?b\ ?k\ 1)\ 0$
using $gap\text{-}Suc(2)$ $block\text{-}Suc(2)$ $assms$ **by** $simp\text{-}all$
then have $x < e\text{-length}\ (block\ i\ (Suc\ t))\ x \neq gap\ i\ (Suc\ t)$
using $that\ length\text{-}block$ **by** $simp\text{-}all$
then have $\tau\ i\ x \downarrow = e\text{-nth}\ (block\ i\ (Suc\ t))\ x$
using $tau\text{-}eq\text{-}state$ **by** $simp$
then show $?thesis$ **using** $that\ assms\ b\ nth\text{-}append$ **by** $(simp\ add: nth\text{-}append)$

next

case $False$

then show $?thesis$ **using** $that\ assms\ tau\text{-}eq\text{-}state'$ **by** $simp$

qed

qed

lemma $change\text{-}on\text{-}1\text{-}hyp\text{-}change:$

assumes $change\text{-}on\text{-}1\ (block\ i\ t)\ (gap\ i\ t)$
shows $\sigma\ i\ ((\tau\ i) \triangleright (t + 1)) \neq \sigma\ i\ ((\tau\ i) \triangleright (gap\ i\ t - 1))$
using $assms\ hd\text{-}block\ init\text{-}tau\text{-}gap\ change\text{-}on\text{-}1\text{-}init\text{-}tau$ **by** $simp$

lemma $change\text{-}on\text{-}either\text{-}hyp\text{-}change:$

assumes $\neg change\text{-}on\text{-}neither\ (block\ i\ t)\ (gap\ i\ t)$
shows $\sigma\ i\ ((\tau\ i) \triangleright (t + 1)) \neq \sigma\ i\ ((\tau\ i) \triangleright (gap\ i\ t - 1))$
using $assms\ change\text{-}on\text{-}0\text{-}hyp\text{-}change\ change\text{-}on\text{-}1\text{-}hyp\text{-}change$ **by** $auto$

lemma $filled\text{-}gap\text{-}0\text{-}init\text{-}tau:$

assumes $f_0 = (\tau\ i)((gap\ i\ t) := Some\ 0)$
shows $f_0 \triangleright (t + 1) = block\ i\ t$
proof $(intro\ initI')$
show $len: e\text{-length}\ (block\ i\ t) = Suc\ (t + 1)$

```

    using assms length-block by auto
  show  $f_0 x \downarrow = e\text{-nth } (block\ i\ t)\ x$  if  $x < Suc\ (t + 1)$  for  $x$ 
  proof (cases  $x = gap\ i\ t$ )
    case True
      then show ?thesis using assms last-block by auto
    next
      case False
        then show ?thesis using assms len tau-eq-state that by auto
  qed
qed

```

lemma *filled-gap-1-init-tau*:

```

  assumes  $f_1 = (\tau\ i)((gap\ i\ t) := Some\ 1)$ 
  shows  $f_1 \triangleright (t + 1) = e\text{-update } (block\ i\ t)\ (gap\ i\ t)\ 1$ 
  proof (intro initI')
    show len:  $e\text{-length } (e\text{-update } (block\ i\ t)\ (gap\ i\ t)\ 1) = Suc\ (t + 1)$ 
      using e-length-update length-block by simp
    show  $f_1 x \downarrow = e\text{-nth } (e\text{-update } (block\ i\ t)\ (gap\ i\ t)\ 1)\ x$  if  $x < Suc\ (t + 1)$  for  $x$ 
    proof (cases  $x = gap\ i\ t$ )
      case True
        moreover have  $gap\ i\ t < e\text{-length } (block\ i\ t)$ 
          using gap-in-block by simp
        ultimately show ?thesis using assms by simp
      next
        case False
          then show ?thesis using assms len tau-eq-state that by auto
    qed
  qed

```

2.9.3 The separating class

Next we define the sets V_i from the introductory proof sketch (page 193).

definition *V-bclim* :: $nat \Rightarrow partial1\ set$ **where**

```

V-bclim i  $\equiv$ 
  if case-two i
  then let  $k = gap\ i\ (LEAST\ t.\ \forall T.\ gap\ i\ T \leq gap\ i\ t)$ 
    in  $\{(\tau\ i)(k := Some\ 0), (\tau\ i)(k := Some\ 1)\}$ 
  else  $\{\tau\ i\}$ 

```

lemma *V-subseteq-R1*: $V\text{-bclim } i \subseteq \mathcal{R}$

```

  proof (cases case-two i)
    case True
      define  $k$  where  $k = gap\ i\ (LEAST\ t.\ \forall T.\ gap\ i\ T \leq gap\ i\ t)$ 
      have  $\tau\ i \in \mathcal{P}$ 
        using tau-in-P2 P2-proj-P1 by auto
      then have  $(\tau\ i)(k := Some\ 0) \in \mathcal{P}$  and  $(\tau\ i)(k := Some\ 1) \in \mathcal{P}$ 
        using P1-update-P1 by simp-all
      moreover have total1  $((\tau\ i)(k := Some\ v))$  for  $v$ 
        using case-two-tau-almost-total(2)[OF True] k-def total1-def by simp
      ultimately have  $(\tau\ i)(k := Some\ 0) \in \mathcal{R}$  and  $(\tau\ i)(k := Some\ 1) \in \mathcal{R}$ 
        using P1-total-imp-R1 by simp-all
      moreover have  $V\text{-bclim } i = \{(\tau\ i)(k := Some\ 0), (\tau\ i)(k := Some\ 1)\}$ 
        using True V-bclim-def k-def by (simp add: Let-def)
      ultimately show ?thesis by simp
    next

```

case *False*
have $V\text{-bclim } i = \{\tau \ i\}$
unfolding $V\text{-bclim-def}$ **by** (*simp add: False*)
moreover have $\tau \ i \in \mathcal{R}$
using *total1I case-one-tau-total[OF False] tau-in-P2 P2-proj-P1[of τ] P1-total-imp-R1*
by *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *case-one-imp-gap-unbounded:*
assumes *case-one i*
shows $\exists t. \text{gap } i \ t - 1 > n$
proof (*induction n*)
case *0*
then show *?case*
using *assms gap-gr0[of i] state-at-0(2)[of i]* **by** (*metis diff-is-0-eq gr-zeroI*)
next
case (*Suc n*)
then obtain t **where** $t: \text{gap } i \ t - 1 > n$
by *auto*
moreover from *assms* **have** $\forall t. \exists T. \text{gap } i \ T > \text{gap } i \ t$
using *leI* **by** *blast*
ultimately obtain T **where** $\text{gap } i \ T > \text{gap } i \ t$
by *auto*
then have $\text{gap } i \ T - 1 > \text{gap } i \ t - 1$
using *gap-gr0[of i]* **by** (*simp add: Suc-le-eq diff-less-mono*)
with t **have** $\text{gap } i \ T - 1 > \text{Suc } n$ **by** *simp*
then show *?case* **by** *auto*
qed

lemma *case-one-imp-not-learn-lim-V:*
assumes *case-one i*
shows $\neg \text{learn-lim } \varphi \ (V\text{-bclim } i) \ (\sigma \ i)$
proof –
have $V\text{-bclim}: V\text{-bclim } i = \{\tau \ i\}$
using *assms V-bclim-def* **by** (*auto simp add: Let-def*)
have $\exists m_1 > n. \exists m_2 > n. (\sigma \ i) ((\tau \ i) \triangleright m_1) \neq (\sigma \ i) ((\tau \ i) \triangleright m_2)$ **for** n
proof –
obtain t **where** $t: \text{gap } i \ t - 1 > n$
using *case-one-imp-gap-unbounded[OF assms]* **by** *auto*
moreover have $\forall t. \exists T \geq t. \neg \text{change-on-neither } (\text{block } i \ T) \ (\text{gap } i \ T)$
using *assms case-two-iff-forever-neither* **by** *blast*
ultimately obtain T **where** $T: T \geq t \wedge \neg \text{change-on-neither } (\text{block } i \ T) \ (\text{gap } i \ T)$
by *auto*
then have $(\sigma \ i) ((\tau \ i) \triangleright (T + 1)) \neq (\sigma \ i) ((\tau \ i) \triangleright (\text{gap } i \ T - 1))$
using *change-on-either-hyp-change* **by** *simp*
moreover have $\text{gap } i \ T - 1 > n$
using $t \ T(1)$ *gap-monotone* **by** (*simp add: diff-le-mono less-le-trans*)
moreover have $T + 1 > n$
proof –
have $\text{gap } i \ T - 1 \leq T$
using *gap-in-block length-block* **by** (*simp add: le-diff-conv less-Suc-eq-le*)
then show *?thesis* **using** $\langle \text{gap } i \ T - 1 > n \rangle$ **by** *simp*
qed
ultimately show *?thesis* **by** *auto*
qed

with *infinite-hyp-changes-not-Lim V-bclim* show *?thesis* by *simp*
qed

lemma *case-two-imp-not-learn-lim-V*:

assumes *case-two i*

shows \neg *learn-lim* φ (*V-bclim i*) (σ *i*)

proof –

let $?P = \lambda t. \forall T. (\text{gap } i \ T) \leq (\text{gap } i \ t)$

let $?t = \text{LEAST } t. ?P \ t$

let $?k = \text{gap } i \ ?t$

let $?b = e\text{-take } ?k \ (\text{block } i \ ?t)$

have $t: \forall T. \text{gap } i \ T \leq \text{gap } i \ ?t$

using *assms LeastI-ex[of ?P]* by *simp*

then have *neither*: $\forall T \geq ?t. \text{change-on-neither } (\text{block } i \ T) \ (\text{gap } i \ T)$

using *gap-le-Suc gap-monotone state-change-on-neither(1)*

by (*metis (no-types, lifting) antisym*)

have *gap-T*: $\forall T \geq ?t. \text{gap } i \ T = ?k$

using *t gap-monotone antisym-conv* by *blast*

define f_0 where $f_0 = (\tau \ i)(?k := \text{Some } 0)$

define f_1 where $f_1 = (\tau \ i)(?k := \text{Some } 1)$

show *?thesis*

proof (*rule same-hyp-for-two-not-Lim*)

show $f_0 \in V\text{-bclim } i$ and $f_1 \in V\text{-bclim } i$

using *assms V-bclim-def f_0-def f_1-def* by (*simp-all add: Let-def*)

show $f_0 \neq f_1$ using *f_0-def f_1-def* by (*meson map-upd-eqD1 zero-neq-one*)

show $\forall n \geq \text{Suc } ?t. \sigma \ i \ (f_0 \triangleright n) = \sigma \ i \ ?b$

proof –

have $\sigma \ i \ (\text{block } i \ T) = \sigma \ i \ (e\text{-take } ?k \ (\text{block } i \ T))$ if $T \geq ?t$ for T

using *that gap-T neither hd-block* by *metis*

then have $\sigma \ i \ (\text{block } i \ T) = \sigma \ i \ ?b$ if $T \geq ?t$ for T

by (*metis (no-types, lifting) init-tau-gap gap-T that*)

then have $\sigma \ i \ (f_0 \triangleright (T + 1)) = \sigma \ i \ ?b$ if $T \geq ?t$ for T

using *filled-gap-0-init-tau[of f_0 i T] f_0-def gap-T that*

by (*metis (no-types, lifting)*)

then have $\sigma \ i \ (f_0 \triangleright T) = \sigma \ i \ ?b$ if $T \geq \text{Suc } ?t$ for T

using *that* by (*metis (no-types, lifting) Suc-eq-plus1 Suc-le-D Suc-le-mono*)

then show *?thesis* by *simp*

qed

show $\forall n \geq \text{Suc } ?t. \sigma \ i \ (f_1 \triangleright n) = \sigma \ i \ ?b$

proof –

have $\sigma \ i \ (e\text{-update } (\text{block } i \ T) \ ?k \ 1) = \sigma \ i \ (e\text{-take } ?k \ (\text{block } i \ T))$ if $T \geq ?t$ for T

using *neither* by (*metis (no-types, lifting) hd-block gap-T that*)

then have $\sigma \ i \ (e\text{-update } (\text{block } i \ T) \ ?k \ 1) = \sigma \ i \ ?b$ if $T \geq ?t$ for T

using *that init-tau-gap[of i] gap-T* by (*metis (no-types, lifting)*)

then have $\sigma \ i \ (f_1 \triangleright (T + 1)) = \sigma \ i \ ?b$ if $T \geq ?t$ for T

using *filled-gap-1-init-tau[of f_1 i T] f_1-def gap-T that*

by (*metis (no-types, lifting)*)

then have $\sigma \ i \ (f_1 \triangleright T) = \sigma \ i \ ?b$ if $T \geq \text{Suc } ?t$ for T

using *that* by (*metis (no-types, lifting) Suc-eq-plus1 Suc-le-D Suc-le-mono*)

then show *?thesis* by *simp*

qed

qed

qed

corollary *not-learn-lim-V*: \neg *learn-lim* φ (*V-bclim i*) (σ *i*)

using *case-one-imp-not-learn-lim-V case-two-imp-not-learn-lim-V*

by (cases case-two i) simp-all

Next we define the separating class.

definition $V\text{-BCLIM} :: \text{partial1 set } (\langle V_{BC-LIM} \rangle)$ **where**
 $V_{BC-LIM} \equiv \bigcup i. V\text{-bclim } i$

lemma $V\text{-BCLIM-R1}: V_{BC-LIM} \subseteq \mathcal{R}$
using $V\text{-BCLIM-def } V\text{-subsetq-R1}$ **by** auto

lemma $V\text{-BCLIM-not-in-Lim}: V_{BC-LIM} \notin LIM$

proof

assume $V_{BC-LIM} \in LIM$
then obtain s **where** $s: \text{learn-lim } \varphi V_{BC-LIM} s$
using $\text{learn-lim-wrt-goedel}[OF \text{ goedel-numbering-phi}] \text{Lim-def}$ **by** blast
moreover obtain i **where** $\varphi i = s$
using $s \text{ learn-limE}(1) \text{ phi-universal}$ **by** blast
ultimately have $\text{learn-lim } \varphi V_{BC-LIM} (\lambda x. \text{eval } r\text{-sigma } [i, x])$
using learn-lim-sigma **by** simp
moreover have $V\text{-bclim } i \subseteq V_{BC-LIM}$
using $V\text{-BCLIM-def}$ **by** auto
ultimately have $\text{learn-lim } \varphi (V\text{-bclim } i) (\lambda x. \text{eval } r\text{-sigma } [i, x])$
using $\text{learn-lim-closed-subsetq}$ **by** simp
then show False
using not-learn-lim-V **by** simp

qed

2.9.4 The separating class is in BC

In order to show $V_{BC-LIM} \in BC$ we define a hypothesis space that for every function τ_i and every list b of numbers contains a copy of τ_i with the first $|b|$ values replaced by b .

definition $\text{psitau} :: \text{partial2 } (\langle \psi^\tau \rangle)$ **where**
 $\psi^\tau b x \equiv (\text{if } x < \text{e-length } b \text{ then Some } (e\text{-nth } b x) \text{ else } \tau (e\text{-hd } b) x)$

lemma $\text{psitau-in-P2}: \psi^\tau \in \mathcal{P}^2$

proof –

define r **where** $r \equiv$
 $Cn \ 2$
 $(r\text{-lifz } r\text{-nth } (Cn \ 2 \ r\text{-tau } [Cn \ 2 \ r\text{-hd } [Id \ 2 \ 0], Id \ 2 \ 1]))$
 $[Cn \ 2 \ r\text{-less } [Id \ 2 \ 1, Cn \ 2 \ r\text{-length } [Id \ 2 \ 0]], Id \ 2 \ 0, Id \ 2 \ 1]$
then have $\text{recfn } 2 \ r$
using $r\text{-tau-recfn}$ **by** simp
moreover have $\text{eval } r [b, x] = \psi^\tau b x$ **for** $b \ x$
proof –
let $?f = Cn \ 2 \ r\text{-tau } [Cn \ 2 \ r\text{-hd } [Id \ 2 \ 0], Id \ 2 \ 1]$
have $\text{recfn } 2 \ r\text{-nth } \text{recfn } 2 \ ?f$
using $r\text{-tau-recfn}$ **by** simp-all
then have $\text{eval } (r\text{-lifz } r\text{-nth } ?f) [c, b, x] =$
 $(\text{if } c = 0 \text{ then eval } r\text{-nth } [b, x] \text{ else eval } ?f [b, x])$ **for** c
by simp
moreover have $\text{eval } r\text{-nth } [b, x] \downarrow = e\text{-nth } b \ x$
by simp
moreover have $\text{eval } ?f [b, x] = \tau (e\text{-hd } b) \ x$
using $r\text{-tau-recfn}$ **by** simp
ultimately have $\text{eval } (r\text{-lifz } r\text{-nth } ?f) [c, b, x] =$

```

      (if c = 0 then Some (e-nth b x) else  $\tau$  (e-hd b) x) for c
    by simp
  moreover have eval (Cn 2 r-less [Id 2 1, Cn 2 r-length [Id 2 0]]) [b, x]  $\downarrow$ =
    (if x < e-length b then 0 else 1)
  by simp
  ultimately show ?thesis
    unfolding r-def psitau-def using r-tau-recfn by simp
qed
ultimately show ?thesis by auto
qed

```

lemma psitau-init:

```

 $\psi^\tau$  (f  $\triangleright$  n) x = (if x < Suc n then Some (the (f x)) else  $\tau$  (the (f 0)) x)
proof -
  let ?e = f  $\triangleright$  n
  have e-length ?e = Suc n by simp
  moreover have x < Suc n  $\implies$  e-nth ?e x = the (f x) by simp
  moreover have e-hd ?e = the (f 0)
    using hd-init by simp
  ultimately show ?thesis using psitau-def by simp
qed

```

The class V_{BC-LIM} can be learned BC-style in the hypothesis space ψ^τ by the identity function.

lemma learn-bc-V-BCLIM: learn-bc ψ^τ V_{BC-LIM} Some

proof (rule learn-bcI)

show environment ψ^τ V_{BC-LIM} Some

using identity-in-R1 V-BCLIM-R1 psitau-in-P2 **by** auto

show $\exists n_0. \forall n \geq n_0. \psi^\tau$ (the (Some (f \triangleright n))) = f **if** $f \in V_{BC-LIM}$ **for** f

proof -

from that V-BCLIM-def **obtain** i **where** i: f \in V-bclim i

by auto

show ?thesis

proof (cases case-two i)

case True

let ?P = $\lambda t. \forall T. (\text{gap } i \ T) \leq (\text{gap } i \ t)$

let ?lmin = LEAST t. ?P t

define k **where** k \equiv gap i ?lmin

have V-bclim: V-bclim i = $\{(\tau \ i)(k := \text{Some } 0), (\tau \ i)(k := \text{Some } 1)\}$

using True V-bclim-def k-def **by** (simp add: Let-def)

moreover have 0 < k

using gap-gr0[of i] k-def **by** simp

ultimately have f 0 \downarrow = i

using tau-at-0[of i] i **by** auto

have ψ^τ (f \triangleright n) = f **if** $n \geq k$ **for** n

proof

fix x

show ψ^τ (f \triangleright n) x = f x

proof (cases x \leq n)

case True

then show ?thesis

using R1-imp-total1 V-subseteq-R1 i psitau-init **by** fastforce

next

case False

then have ψ^τ (f \triangleright n) x = τ (the (f 0)) x

using psitau-init **by** simp

```

    then have  $\psi^\tau (f \triangleright n) x = \tau i x$ 
      using  $\langle f 0 \Downarrow = i \rangle$  by simp
    moreover have  $f x = \tau i x$ 
      using False V-bclim i that by auto
    ultimately show ?thesis by simp
  qed
qed
then show ?thesis by auto
next
case False
then have  $V\text{-bclim } i = \{\tau i\}$ 
  using V-bclim-def by (auto simp add: Let-def)
then have  $f: f = \tau i$ 
  using i by simp
have  $\psi^\tau (f \triangleright n) = f$  for  $n$ 
proof
  fix  $x$ 
  show  $\psi^\tau (f \triangleright n) x = f x$ 
  proof (cases  $x \leq n$ )
    case True
    then show ?thesis
      using R1-imp-total1 V-BCLIM-R1 psitau-init that by auto
  next
    case False
    then show ?thesis by (simp add: f psitau-init tau-at-0)
  qed
qed
then show ?thesis by simp
qed
qed
qed

```

Finally, the main result of this section:

theorem *Lim-subset-BC*: $LIM \subset BC$
 using *learn-bc V-BCLIM BC-def Lim-subseteq-BC V-BCLIM-not-in-Lim* by auto

end

2.10 TOTAL is a proper subset of CONS

```

theory TOTAL-CONS
  imports Lemma-R
          CP-FIN-NUM
          CONS-LIM
begin

```

We first show that TOTAL is a subset of CONS. Then we present a separating class.

2.10.1 TOTAL is a subset of CONS

A TOTAL strategy hypothesizes only total functions, for which the consistency with the input prefix is decidable. A CONS strategy can thus run a TOTAL strategy and check if its hypothesis is consistent. If so, it outputs this hypothesis, otherwise some arbitrary consistent one. Since the TOTAL strategy converges to a correct hypothesis, which is consistent, the CONS strategy will converge to the same hypothesis.

Without loss of generality we can assume that learning takes place with respect to our Gödel numbering φ . So we need to decide consistency only for this numbering.

abbreviation *r-consist-phi* **where**
r-consist-phi \equiv *r-consistent r-phi*

lemma *r-consist-phi-recfn* [*simp*]: *recfn 2 r-consist-phi*
by *simp*

lemma *r-consist-phi*:
assumes $\forall k < e\text{-length } e. \varphi \ i \ k \downarrow$
shows *eval r-consist-phi* [*i*, *e*] $\downarrow =$
(if $\forall k < e\text{-length } e. \varphi \ i \ k \downarrow = e\text{-nth } e \ k$ then 0 else 1)

proof –
have $\forall k < e\text{-length } e. \text{eval } r\text{-phi} \ [i, k] \downarrow$
using *assms phi-def* **by** *simp*
moreover have *recfn 2 r-phi* **by** *simp*
ultimately have *eval (r-consistent r-phi)* [*i*, *e*] $\downarrow =$
(if $\forall k < e\text{-length } e. \text{eval } r\text{-phi} \ [i, k] \downarrow = e\text{-nth } e \ k$ then 0 else 1)
using *r-consistent-converg assms* **by** *simp*
then show *?thesis* **using** *phi-def* **by** *simp*
qed

lemma *r-consist-phi-init*:
assumes $f \in \mathcal{R}$ **and** $\varphi \ i \in \mathcal{R}$
shows *eval r-consist-phi* [*i*, $f \triangleright n$] $\downarrow =$ *(if $\forall k \leq n. \varphi \ i \ k = f \ k$ then 0 else 1)*
using *assms r-consist-phi R1-imp-total1 total1E* **by** (*simp add: r-consist-phi*)

lemma *TOTAL-subseteq-CONS*: *TOTAL* \subseteq *CONS*
proof

fix *U* **assume** $U \in \text{TOTAL}$
then have $U \in \text{TOTAL-wrt } \varphi$
using *TOTAL-wrt-phi-eq-TOTAL* **by** *blast*
then obtain *t'* **where** *t': learn-total $\varphi \ U \ t'$*
using *TOTAL-wrt-def* **by** *auto*
then obtain *t* **where** *t: recfn 1 t $\wedge x. \text{eval } t \ [x] = t' \ x$*
using *learn-totalE(1) PIE* **by** *blast*
then have *t-converg: eval t [f \triangleright n] \downarrow if $f \in U$ for $f \ n$*
using *t' learn-totalE(1) that* **by** *auto*

define *s* **where** $s \equiv Cn \ 1 \ r\text{-ifz} \ [Cn \ 1 \ r\text{-consist-phi} \ [t, Id \ 1 \ 0], t, r\text{-auxhyp}]$
then have *recfn 1 s*
using *r-consist-phi-recfn r-auxhyp-prim t(1)* **by** *simp*

have *consist: eval r-consist-phi [the (eval t [f \triangleright n]), f \triangleright n] $\downarrow =$*
(if $\forall k \leq n. \varphi \ (\text{the } (\text{eval } t \ [f \ \triangleright \ n])) \ k = f \ k$ then 0 else 1)
if $f \in U$ **for** $f \ n$

proof –
have *eval r-consist-phi [the (eval t [f \triangleright n]), f \triangleright n] =*
eval (Cn 1 r-consist-phi [t, Id 1 0]) [f \triangleright n]
using *that t-converg t(1)* **by** *simp*
also have $\dots \downarrow =$ *(if $\forall k \leq n. \varphi \ (\text{the } (\text{eval } t \ [f \ \triangleright \ n])) \ k = f \ k$ then 0 else 1)*
proof –
from that have $f \in \mathcal{R}$
using *learn-totalE(1) t'* **by** *blast*
moreover have $\varphi \ (\text{the } (\text{eval } t \ [f \ \triangleright \ n])) \in \mathcal{R}$

using $t' t$ learn-totalE t -converg that by simp
 ultimately show ?thesis
 using r -consist-phi-init t -converg $t(1)$ that by simp
 qed
 finally show ?thesis .
 qed

have s -eq- t : eval $s [f \triangleright n] = \text{eval } t [f \triangleright n]$
 if $\forall k \leq n. \varphi (\text{the } (\text{eval } t [f \triangleright n])) k = f k$ and $f \in U$ for $f n$
 using that consist s -def t r -auxhyp-prim prim-recfn-total
 by simp

have s -eq-aux: eval $s [f \triangleright n] = \text{eval } r\text{-auxhyp } [f \triangleright n]$
 if $\neg (\forall k \leq n. \varphi (\text{the } (\text{eval } t [f \triangleright n])) k = f k)$ and $f \in U$ for $f n$
 proof –
 from that have eval r -consist-phi $[\text{the } (\text{eval } t [f \triangleright n]), f \triangleright n] \downarrow = 1$
 using consist by simp
 moreover have $t' (f \triangleright n) \downarrow$ using $t' t$ learn-totalE(1) that(2) by blast
 ultimately show ?thesis
 using s -def t r -auxhyp-prim $t' t$ learn-totalE by simp
 qed

have learn-cons $\varphi U (\lambda e. \text{eval } s [e])$
 proof (rule learn-consI)
 have eval $s [f \triangleright n] \downarrow$ if $f \in U$ for $f n$
 using that t -converg[OF that, of n] s -eq- t [of $n f$] prim-recfn-total[of r -auxhyp 1]
 r -auxhyp-prim s -eq-aux[OF - that, of n] totalE
 by fastforce
 then show environment $\varphi U (\lambda e. \text{eval } s [e])$
 using $t' \langle \text{recfn } 1 s \rangle$ learn-totalE(1) by blast
 show $\exists i. \varphi i = f \wedge (\forall^\infty n. \text{eval } s [f \triangleright n] \downarrow = i)$ if $f \in U$ for f
 proof –
 from that $t' t$ learn-totalE obtain $i n_0$ where
 $i\text{-}n0$: $\varphi i = f \wedge (\forall n \geq n_0. \text{eval } t [f \triangleright n] \downarrow = i)$
 by metis
 then have $\bigwedge n. n \geq n_0 \implies \forall k \leq n. \varphi (\text{the } (\text{eval } t [f \triangleright n])) k = f k$
 by simp
 with s -eq- t have $\bigwedge n. n \geq n_0 \implies \text{eval } s [f \triangleright n] = \text{eval } t [f \triangleright n]$
 using that by simp
 with $i\text{-}n0$ have $\bigwedge n. n \geq n_0 \implies \text{eval } s [f \triangleright n] \downarrow = i$
 by auto
 with $i\text{-}n0$ show ?thesis by auto
 qed
 show $\forall k \leq n. \varphi (\text{the } (\text{eval } s [f \triangleright n])) k = f k$ if $f \in U$ for $f n$
 proof (cases $\forall k \leq n. \varphi (\text{the } (\text{eval } t [f \triangleright n])) k = f k$)
 case True
 with that s -eq- t show ?thesis by simp
 next
 case False
 then have eval $s [f \triangleright n] = \text{eval } r\text{-auxhyp } [f \triangleright n]$
 using that s -eq-aux by simp
 moreover have $f \in \mathcal{R}$
 using learn-totalE(1)[OF t'] that by auto
 ultimately show ?thesis using r -auxhyp by simp
 qed
 qed

then show $U \in \text{CONS}$ using *CONS-def* by *auto*
qed

2.10.2 The separating class

Definition of the class

The class that will be shown to be in $\text{CONS} - \text{TOTAL}$ is the union of the following two classes.

definition *V-constotal-1* :: *partial1 set* **where**

$$V\text{-constotal-1} \equiv \{f. \exists j p. f = [j] \odot p \wedge j \geq 2 \wedge p \in \mathcal{R}_{01} \wedge \varphi j = f\}$$

definition *V-constotal-2* :: *partial1 set* **where**

$$\begin{aligned} V\text{-constotal-2} \equiv & \{f. \exists j a k. \\ & f = j \# a @ [k] \odot 0^\infty \wedge \\ & j \geq 2 \wedge \\ & (\forall i < \text{length } a. a ! i \leq 1) \wedge \\ & k \geq 2 \wedge \\ & \varphi j = j \# a \odot \uparrow^\infty \wedge \\ & \varphi k = f\} \end{aligned}$$

definition *V-constotal* :: *partial1 set* **where**

$$V\text{-constotal} \equiv V\text{-constotal-1} \cup V\text{-constotal-2}$$

lemma *V-constotal-2I*:

assumes $f = j \# a @ [k] \odot 0^\infty$
and $j \geq 2$
and $\forall i < \text{length } a. a ! i \leq 1$
and $k \geq 2$
and $\varphi j = j \# a \odot \uparrow^\infty$
and $\varphi k = f$
shows $f \in V\text{-constotal-2}$
using *assms V-constotal-2-def* by *blast*

lemma *V-subseteq-R1*: $V\text{-constotal} \subseteq \mathcal{R}$

proof

fix f **assume** $f \in V\text{-constotal}$
then have $f \in V\text{-constotal-1} \vee f \in V\text{-constotal-2}$
using *V-constotal-def* by *auto*
then show $f \in \mathcal{R}$
proof
assume $f \in V\text{-constotal-1}$
then obtain $j p$ **where** $f = [j] \odot p$ $p \in \mathcal{R}_{01}$
using *V-constotal-1-def* by *blast*
then show *?thesis* **using** *prepend-in-R1 RPred1-subseteq-R1* by *auto*
next
assume $f \in V\text{-constotal-2}$
then obtain $j a k$ **where** $f = j \# a @ [k] \odot 0^\infty$
using *V-constotal-2-def* by *blast*
then show *?thesis* **using** *almost0-in-R1* by *auto*
qed
qed

The class is in CONS

The class can be learned by the strategy *rmge2*, which outputs the rightmost value greater or equal two in the input f^n . If f is from V_1 then the strategy is correct right from the start. If f is from V_2 the strategy outputs the consistent hypothesis j until it encounters the correct hypothesis k , to which it converges.

lemma *V-in-CONS: learn-cons* φ *V-constotal* *rmge2*

proof (*rule learn-consI*)

show *environment* φ *V-constotal* *rmge2*

using *V-subseteq-R1* *rmge2-in-R1* *R1-imp-total1* *phi-in-P2* **by** *simp*

have $(\exists i. \varphi i = f \wedge (\forall^\infty n. \text{rmge2}(f \triangleright n) \downarrow = i)) \wedge$
 $(\forall n. \forall k \leq n. \varphi(\text{the}(\text{rmge2}(f \triangleright n))) k = f k)$

if $f \in V\text{-constotal}$ **for** f

proof (*cases* $f \in V\text{-constotal-1}$)

case *True*

then obtain j p **where**

$f: f = [j] \odot p$ **and**

$j: j \geq 2$ **and**

$p: p \in \mathcal{R}_{01}$ **and**

phi-j: $\varphi j = f$

using *V-constotal-1-def* **by** *blast*

then have $f 0 \downarrow = j$ **by** (*simp add: prepend-at-less*)

then have *f-at-0*: *the* $(f 0) \geq 2$ **by** (*simp add: j*)

have *f-at-gr0*: *the* $(f x) \leq 1$ **if** $x > 0$ **for** x

using *that f p* **by** (*simp add: RPred1-altdef Suc-leI prepend-at-ge*)

have *total1* f

using *V-subseteq-R1* *that R1-imp-total1* *total1-def* **by** *auto*

have $\text{rmge2}(f \triangleright n) \downarrow = j$ **for** n

proof –

let $?P = \lambda i. i < \text{Suc } n \wedge \text{the}(f i) \geq 2$

have *Greatest* $?P = 0$

proof (*rule Greatest-equality*)

show $0 < \text{Suc } n \wedge 2 \leq \text{the}(f 0)$

using *f-at-0* **by** *simp*

show $\bigwedge y. y < \text{Suc } n \wedge 2 \leq \text{the}(f y) \implies y \leq 0$

using *f-at-gr0* **by** *fastforce*

qed

then have $\text{rmge2}(f \triangleright n) = f 0$

using *f-at-0* *rmge2-init-total*[*of f n, OF* $\langle \text{total1 } f \rangle$] **by** *auto*

then show $\text{rmge2}(f \triangleright n) \downarrow = j$

by (*simp add: f 0* $\downarrow = j$)

qed

then show *?thesis* **using** *phi-j* **by** *auto*

next

case *False*

then have $f \in V\text{-constotal-2}$

using *V-constotal-def* *that* **by** *auto*

then obtain j a k **where** *jak*:

$f = j \# a @ [k] \odot 0^\infty$

$j \geq 2$

$\forall i < \text{length } a. a ! i \leq 1$

$k \geq 2$

$\varphi j = j \# a \odot \uparrow^\infty$

$\varphi k = f$

using *V-constotal-2-def* **by** *blast*

then have $f\text{-at-}0$: $f\ 0 \Downarrow = j$ **by** *simp*
have $f\text{-eq-}a$: $f\ x \Downarrow = a \wedge (x - 1) \text{ if } 0 < x \wedge x < \text{Suc } (\text{length } a)$ **for** x
proof –
 have $x - 1 < \text{length } a$
 using *that* **by** *auto*
 then show *?thesis*
 by (*simp add: jak(1) less-SucI nth-append that*)
qed
then have $f\text{-at-}a$: *the* $(f\ x) \leq 1$ **if** $0 < x \wedge x < \text{Suc } (\text{length } a)$ **for** x
 using *jak(3) that* **by** *auto*
from *jak* **have** $f\text{-k}$: $f\ (\text{Suc } (\text{length } a)) \Downarrow = k$ **by** *auto*
from *jak* **have** $f\text{-at-big}$: $f\ x \Downarrow = 0$ **if** $x > \text{Suc } (\text{length } a)$ **for** x
 using *that* **by** *simp*
let $?P = \lambda n\ i. i < \text{Suc } n \wedge \text{the } (f\ i) \geq 2$
have *rmge2*: $\text{rmge2 } (f\ \triangleright\ n) = f\ (\text{Greatest } (?P\ n))$ **for** n
proof –
 have $\neg (\forall i < \text{Suc } n. \text{the } (f\ i) < 2)$ **for** n
 using *jak(2) f-at-0* **by** *auto*
 moreover have *total1 f*
 using *V-subseteq-R1 R1-imp-total1 that total1-def* **by** *auto*
 ultimately show *?thesis* **using** *rmge2-init-total[of f n]* **by** *auto*
qed
have $\text{Greatest } (?P\ n) = 0$ **if** $n < \text{Suc } (\text{length } a)$ **for** n
proof (*rule Greatest-equality*)
 show $0 < \text{Suc } n \wedge 2 \leq \text{the } (f\ 0)$
 using *that* **by** (*simp add: jak(2) f-at-0*)
 show $\bigwedge y. y < \text{Suc } n \wedge 2 \leq \text{the } (f\ y) \implies y \leq 0$
 using *that f-at-a*
 by (*metis Suc-1 dual-order.strict-trans leI less-Suc-eq not-less-eq-eq*)
qed
with *rmge2 f-at-0* **have** *rmge2-small*:
 $\text{rmge2 } (f\ \triangleright\ n) \Downarrow = j$ **if** $n < \text{Suc } (\text{length } a)$ **for** n
 using *that* **by** *simp*
have $\text{Greatest } (?P\ n) = \text{Suc } (\text{length } a)$ **if** $n \geq \text{Suc } (\text{length } a)$ **for** n
proof (*rule Greatest-equality*)
 show $\text{Suc } (\text{length } a) < \text{Suc } n \wedge 2 \leq \text{the } (f\ (\text{Suc } (\text{length } a)))$
 using *that f-k* **by** (*simp add: jak(4) less-Suc-eq-le*)
 show $\bigwedge y. y < \text{Suc } n \wedge 2 \leq \text{the } (f\ y) \implies y \leq \text{Suc } (\text{length } a)$
 using *that f-at-big* **by** (*metis leI le-SucI not-less-eq-eq numeral-2-eq-2 option.sel*)
qed
with *rmge2 f-at-big f-k* **have** *rmge2-big*:
 $\text{rmge2 } (f\ \triangleright\ n) \Downarrow = k$ **if** $n \geq \text{Suc } (\text{length } a)$ **for** n
 using *that* **by** *simp*
then have $\exists i\ n_0. \varphi\ i = f \wedge (\forall n \geq n_0. \text{rmge2 } (f\ \triangleright\ n) \Downarrow = i)$
 using *jak(6)* **by** *auto*
moreover have $\forall k \leq n. \varphi\ (\text{the } (\text{rmge2 } (f\ \triangleright\ n)))\ k = f\ k$ **for** n
proof (*cases n < Suc (length a)*)
 case *True*
 then have $\text{rmge2 } (f\ \triangleright\ n) \Downarrow = j$
 using *rmge2-small* **by** *simp*
 then have $\varphi\ (\text{the } (\text{rmge2 } (f\ \triangleright\ n))) = \varphi\ j$ **by** *simp*
 with *True* **show** *?thesis*
 using *rmge2-small f-at-0 f-eq-a jak(5) prepend-at-less*
 by (*metis le-less-trans le-zero-eq length-Cons not-le-imp-less nth-Cons-0 nth-Cons-pos*)
next
 case *False*

then show *?thesis using rmge2-big jak by simp*
qed
ultimately show *?thesis by simp*
qed
then show $\bigwedge f. f \in V\text{-constotal} \implies \exists i. \varphi i = f \wedge (\forall^\infty n. \text{rmge2 } (f \triangleright n) \downarrow = i)$
and $\bigwedge f n. f \in V\text{-constotal} \implies \forall k \leq n. \varphi (\text{the } (\text{rmge2 } (f \triangleright n))) k = f k$
by simp-all
qed

The class is not in TOTAL

Recall that V is the union of $V_1 = \{jp \mid j \geq 2 \wedge p \in \mathcal{R}_{01} \wedge \varphi_j = jp\}$ and $V_2 = \{jak0^\infty \mid j \geq 2 \wedge a \in \{0,1\}^* \wedge k \geq 2 \wedge \varphi_j = ja \uparrow^\infty \wedge \varphi_k = jak0^\infty\}$.

The proof is adapted from a proof of a stronger result by Freivalds, Kinber, and Wiehagen [7, Theorem 27] concerning an inference type not defined here.

The proof is by contradiction. If V was in TOTAL, there would be a strategy S learning V in our standard Gödel numbering φ . By Lemma R for TOTAL we can assume S to be total.

In order to construct a function $f \in V$ for which S fails we employ a computable process iteratively building function prefixes. For every j the process builds a function ψ_j . The initial prefix is the singleton $[j]$. Given a prefix b , the next prefix is determined as follows:

1. Search for a $y \geq |b|$ with $\varphi_{S(b)}(y) \downarrow = v$ for some v .
2. Set the new prefix $b0^{y-|b|}\bar{v}$, where $\bar{v} = 1 - v$.

Step 1 can diverge, for example, if $\varphi_{S(b)}$ is the empty function. In this case ψ_j will only be defined for a finite prefix. If, however, Step 2 is reached, the prefix b is extended to a b' such that $\varphi_{S(b)}(y) \neq b'_y$, which implies $S(b)$ is a wrong hypothesis for every function starting with b' , in particular for ψ_j . Since $\bar{v} \in \{0,1\}$, Step 2 only appends zeros and ones, which is important for showing membership in V .

This process defines a numbering $\psi \in \mathcal{P}^2$, and by Kleene's fixed-point theorem there is a $j \geq 2$ with $\varphi_j = \psi_j$. For this j there are two cases:

- Case 1. Step 1 always succeeds. Then ψ_j is total and $\psi_j \in V_1$. But S outputs wrong hypotheses on infinitely many prefixes of ψ_j (namely every prefix constructed by the process).
- Case 2. Step 1 diverges at some iteration, say when the state is $b = ja$ for some $a \in \{0,1\}^*$. Then ψ_j has the form $ja \uparrow^\infty$. The numbering χ with $\chi_k = jak0^\infty$ is in \mathcal{P}^2 , and by Kleene's fixed-point theorem there is a $k \geq 2$ with $\varphi_k = \chi_k = jak0^\infty$. This $jak0^\infty$ is in V_2 and has the prefix ja . But Step 1 diverged on this prefix, which means there is no $y \geq |ja|$ with $\varphi_{S(ja)}(y) \downarrow$. In other words S hypothesizes a non-total function.

Thus, in both cases there is a function in V where S does not behave like a TOTAL strategy. This is the desired contradiction.

The following locale formalizes this proof sketch.

locale *total-cons* =
fixes $s :: \text{partial1}$
assumes *s-in-R1*: $s \in \mathcal{R}$

begin

definition $r\text{-s} :: \text{recf}$ **where**

$r\text{-s} \equiv \text{SOME } r\text{-s. recfn } 1 \text{ } r\text{-s} \wedge \text{total } r\text{-s} \wedge s = (\lambda x. \text{eval } r\text{-s } [x])$

lemma $rs\text{-recfn}$ $[simp]: \text{recfn } 1 \text{ } r\text{-s}$

and $rs\text{-total}$ $[simp]: \bigwedge x. \text{eval } r\text{-s } [x] \downarrow$

and $eval\text{-rs}: \bigwedge x. s \ x = \text{eval } r\text{-s } [x]$

using $r\text{-s-def}$ $R1\text{-SOME}[OF \ s\text{-in-}R1, \text{ of } r\text{-s}]$ **by** $simp\text{-all}$

Performing Step 1 means enumerating the domain of $\varphi_{S(b)}$ until a $y \geq |b|$ is found. The next function enumerates all domain values and checks the condition for them.

definition $r\text{-search-enum} \equiv$

$Cn \ 2 \ r\text{-le } [Cn \ 2 \ r\text{-length } [Id \ 2 \ 1], Cn \ 2 \ r\text{-enumdom } [Cn \ 2 \ r\text{-s } [Id \ 2 \ 1], Id \ 2 \ 0]]$

lemma $r\text{-search-enum-recfn}$ $[simp]: \text{recfn } 2 \ r\text{-search-enum}$

by $(simp \text{ add: } r\text{-search-enum-def} \text{ Let-def})$

abbreviation $search\text{-enum} :: \text{partial2}$ **where**

$search\text{-enum } x \ b \equiv \text{eval } r\text{-search-enum } [x, b]$

abbreviation $enumdom :: \text{partial2}$ **where**

$enumdom \ i \ y \equiv \text{eval } r\text{-enumdom } [i, y]$

lemma $enumdom\text{-empty-domain}$:

assumes $\bigwedge x. \varphi \ i \ x \uparrow$

shows $\bigwedge y. \text{enumdom } i \ y \uparrow$

using $assms \ r\text{-enumdom-empty-domain}$ **by** $(simp \text{ add: } \text{phi-def})$

lemma $enumdom\text{-nonempty-domain}$:

assumes $\varphi \ i \ x_0 \downarrow$

shows $\bigwedge y. \text{enumdom } i \ y \downarrow$

and $\bigwedge x. \varphi \ i \ x \downarrow \longleftrightarrow (\exists y. \text{enumdom } i \ y \downarrow = x)$

using $assms \ r\text{-enumdom-nonempty-domain} \ \text{phi-def}$ **by** metis+

Enumerating the empty domain yields the empty function.

lemma $search\text{-enum-empty}$:

fixes $b :: \text{nat}$

assumes $s \ b \downarrow = i$ **and** $\bigwedge x. \varphi \ i \ x \uparrow$

shows $\bigwedge x. \text{search-enum } x \ b \uparrow$

using $assms \ r\text{-search-enum-def} \ \text{enumdom-empty-domain} \ \text{eval-rs}$ **by** $simp$

Enumerating a non-empty domain yields a total function.

lemma $search\text{-enum-nonempty}$:

fixes $b \ y_0 :: \text{nat}$

assumes $s \ b \downarrow = i$ **and** $\varphi \ i \ y_0 \downarrow$ **and** $e = \text{the } (\text{enumdom } i \ x)$

shows $\text{search-enum } x \ b \downarrow = (\text{if } e\text{-length } b \leq e \text{ then } 0 \text{ else } 1)$

proof –

let $?e = \lambda x. \text{the } (\text{enumdom } i \ x)$

let $?y = Cn \ 2 \ r\text{-enumdom } [Cn \ 2 \ r\text{-s } [Id \ 2 \ 1], Id \ 2 \ 0]$

have $\text{recfn } 2 \ ?y$ **using** $assms(1)$ **by** $simp$

moreover have $\bigwedge x. \text{eval } ?y \ [x, b] = \text{enumdom } i \ x$

using $assms(1,2) \ \text{eval-rs}$ **by** $auto$

moreover from this have $\bigwedge x. \text{eval } ?y \ [x, b] \downarrow$

using $\text{enumdom-nonempty-domain}(1)[OF \ \text{assms}(2)]$ **by** $simp$

ultimately have $eval (Cn\ 2\ r-le\ [Cn\ 2\ r-length\ [Id\ 2\ 1],\ ?y])\ [x,\ b]\ \Downarrow =$
 $(if\ e-length\ b\ \leq\ ?e\ x\ then\ 0\ else\ 1)$
by *simp*
then show *?thesis* **using** *assms* **by** (*simp* *add: r-search-enum-def*)
qed

If there is a y as desired, the enumeration will eventually return zero (representing “true”).

lemma *search-enum-nonempty-eq0*:
fixes $b\ y :: nat$
assumes $s\ b\ \Downarrow = i$ **and** $\varphi\ i\ y\ \Downarrow$ **and** $y \geq e-length\ b$
shows $\exists x. search-enum\ x\ b\ \Downarrow = 0$
proof –
obtain x **where** $x: enumdom\ i\ x\ \Downarrow = y$
using *enumdom-nonempty-domain(2)* [*OF* *assms(2)*] *assms(2)* **by** *auto*
from *assms(2)* **have** $\varphi\ i\ y\ \Downarrow$ **by** *simp*
with x **have** $search-enum\ x\ b\ \Downarrow = 0$
using *search-enum-nonempty* [**where** *?e=y*] *assms* **by** *auto*
then show *?thesis* **by** *auto*
qed

If there is no y as desired, the enumeration will never return zero.

lemma *search-enum-nonempty-neq0*:
fixes $b\ y0 :: nat$
assumes $s\ b\ \Downarrow = i$
and $\varphi\ i\ y0\ \Downarrow$
and $\neg (\exists y. \varphi\ i\ y\ \Downarrow \wedge y \geq e-length\ b)$
shows $\neg (\exists x. search-enum\ x\ b\ \Downarrow = 0)$
proof
assume $\exists x. search-enum\ x\ b\ \Downarrow = 0$
then obtain x **where** $x: search-enum\ x\ b\ \Downarrow = 0$
by *auto*
obtain y **where** $y: enumdom\ i\ x\ \Downarrow = y$
using *enumdom-nonempty-domain* [*OF* *assms(2)*] **by** *blast*
then have $search-enum\ x\ b\ \Downarrow = (if\ e-length\ b\ \leq\ y\ then\ 0\ else\ 1)$
using *assms(1-2)* *search-enum-nonempty* **by** *simp*
with x **have** $e-length\ b \leq y$
using *option.inject* **by** *fastforce*
moreover have $\varphi\ i\ y\ \Downarrow$
using *assms(2)* *enumdom-nonempty-domain(2)* y **by** *blast*
ultimately show *False* **using** *assms(3)* **by** *force*
qed

The next function corresponds to Step 1. Given a prefix b it computes a $y \geq |b|$ with $\varphi_{S(b)}(y) \Downarrow$ if such a y exists; otherwise it diverges.

definition *r-search* $\equiv Cn\ 1\ r-enumdom\ [r-s,\ Mn\ 1\ r-search-enum]$

lemma *r-search-recfn* [*simp*]: *recfn 1 r-search*
using *r-search-def* **by** *simp*

abbreviation *search* $:: partial1$ **where**
 $search\ b \equiv eval\ r-search\ [b]$

If $\varphi_{S(b)}$ is the empty function, the search process diverges because already the enumeration of the domain diverges.

lemma *search-empty*:
assumes $s\ b \downarrow = i$ **and** $\bigwedge x. \varphi\ i\ x \uparrow$
shows $\text{search}\ b \uparrow$
proof –
have $\bigwedge x. \text{search-enum}\ x\ b \uparrow$
using *search-enum-empty*[*OF assms*] **by** *simp*
then have $\text{eval}\ (Mn\ 1\ r\text{-search-enum})\ [b] \uparrow$ **by** *simp*
then show $\text{search}\ b \uparrow$ **unfolding** *r-search-def* **by** *simp*
qed

If $\varphi_{S(b)}$ is non-empty, but there is no y with the desired properties, the search process diverges.

lemma *search-nonempty-neq0*:
fixes $b\ y_0 :: \text{nat}$
assumes $s\ b \downarrow = i$
and $\varphi\ i\ y_0 \downarrow$
and $\neg (\exists y. \varphi\ i\ y \downarrow \wedge y \geq e\text{-length}\ b)$
shows $\text{search}\ b \uparrow$
proof –
have $\neg (\exists x. \text{search-enum}\ x\ b \downarrow = 0)$
using *assms search-enum-nonempty-neq0* **by** *simp*
moreover have *recfn 1 (Mn 1 r-search-enum)*
by (*simp add: assms(1)*)
ultimately have $\text{eval}\ (Mn\ 1\ r\text{-search-enum})\ [b] \uparrow$ **by** *simp*
then show *?thesis* **using** *r-search-def* **by** *auto*
qed

If there is a y as desired, the search process will return one such y .

lemma *search-nonempty-eq0*:
fixes $b\ y :: \text{nat}$
assumes $s\ b \downarrow = i$ **and** $\varphi\ i\ y \downarrow$ **and** $y \geq e\text{-length}\ b$
shows $\text{search}\ b \downarrow$
and $\varphi\ i\ (\text{the}\ (\text{search}\ b)) \downarrow$
and $\text{the}\ (\text{search}\ b) \geq e\text{-length}\ b$
proof –
have $\exists x. \text{search-enum}\ x\ b \downarrow = 0$
using *assms search-enum-nonempty-eq0* **by** *simp*
moreover have $\forall x. \text{search-enum}\ x\ b \downarrow$
using *assms search-enum-nonempty* **by** *simp*
moreover have *recfn 1 (Mn 1 r-search-enum)*
by *simp*
ultimately have
1: $\text{search-enum}\ (\text{the}\ (\text{eval}\ (Mn\ 1\ r\text{-search-enum})\ [b]))\ b \downarrow = 0$ **and**
2: $\text{eval}\ (Mn\ 1\ r\text{-search-enum})\ [b] \downarrow$
using *eval-Mn-diverg eval-Mn-convergeE*[*of 1 r-search-enum [b]*]
by (*metis (no-types, lifting) One-nat-def length-Cons list.size(3) option.collapse,*
metis (no-types, lifting) One-nat-def length-Cons list.size(3))
let $?x = \text{the}\ (\text{eval}\ (Mn\ 1\ r\text{-search-enum})\ [b])$
have $\text{search}\ b = \text{eval}\ (Cn\ 1\ r\text{-enumdom}\ [r\text{-s}, Mn\ 1\ r\text{-search-enum}])\ [b]$
unfolding *r-search-def* **by** *simp*
then have $?3: \text{search}\ b = \text{enumdom}\ i\ ?x$
using *assms 2 eval-rs* **by** *simp*
then have $\text{the}\ (\text{search}\ b) = \text{the}\ (\text{enumdom}\ i\ ?x)$ **(is** $?y = -$
by *simp*
then have $?4: \text{search-enum}\ ?x\ b \downarrow = (\text{if}\ e\text{-length}\ b \leq ?y\ \text{then}\ 0\ \text{else}\ 1)$

```

    using search-enum-nonempty assms by simp
  from 3 have  $\varphi i ?y \downarrow$ 
    using enumdom-nonempty-domain assms(2) by (metis option.collapse)
  then show  $\varphi i ?y \downarrow$ 
    using phi-def by simp
  then show  $?y \geq e\text{-length } b$ 
    using assms 4 1 option.inject by fastforce
  show search b  $\downarrow$ 
    using 3 assms(2) enumdom-nonempty-domain(1) by auto
qed

```

The converse of the previous lemma states that whenever the search process returns a value it will be one with the desired properties.

```

lemma search-converg:
  assumes  $s b \downarrow = i$  and search b  $\downarrow$  (is  $?y \downarrow$ )
  shows  $\varphi i$  (the  $?y$ )  $\downarrow$ 
    and the  $?y \geq e\text{-length } b$ 
proof -
  have  $\exists y. \varphi i y \downarrow$ 
    using assms search-empty by meson
  then have  $\exists y. y \geq e\text{-length } b \wedge \varphi i y \downarrow$ 
    using search-nonempty-neq0 assms by meson
  then obtain y where  $y \geq e\text{-length } b \wedge \varphi i y \downarrow$  by auto
  then have  $\varphi i y \downarrow$ 
    using phi-def by simp
  then show  $\varphi i$  (the (search b))  $\downarrow$ 
    and (the (search b))  $\geq e\text{-length } b$ 
    using y assms search-nonempty-eq0[OF assms(1)  $\langle \varphi i y \downarrow \rangle$ ] by simp-all
qed

```

Likewise, if the search diverges, there is no appropriate y .

```

lemma search-diverg:
  assumes  $s b \downarrow = i$  and search b  $\uparrow$ 
  shows  $\neg (\exists y. \varphi i y \downarrow \wedge y \geq e\text{-length } b)$ 
proof
  assume  $\exists y. \varphi i y \downarrow \wedge y \geq e\text{-length } b$ 
  then obtain y where  $y: \varphi i y \downarrow \wedge y \geq e\text{-length } b$ 
    by auto
  from y(1) have  $\varphi i y \downarrow$ 
    by (simp add: phi-def)
  with y(2) search-nonempty-eq0 have search b  $\downarrow$ 
    using assms by blast
  with assms(2) show False by simp
qed

```

Step 2 extends the prefix by a block of the shape $0^n \bar{v}$. The next function constructs such a block for given n and v .

```

definition r-badblock  $\equiv$ 
  let  $f = Cn\ 1\ r\text{-singleton-encode } [r\text{-not}]$ ;
     $g = Cn\ 3\ r\text{-cons } [r\text{-constn } 2\ 0, Id\ 3\ 1]$ 
  in  $Pr\ 1\ f\ g$ 

```

```

lemma r-badblock-prim [simp]:  $recfn\ 2\ r\text{-badblock}$ 
  unfolding r-badblock-def by simp

```

lemma *r-badblock*: $\text{eval } r\text{-badblock } [n, v] \Downarrow = \text{list-encode } (\text{replicate } n \ 0 \ @ \ [1 - v])$

proof (*induction n*)

case *0*

let $?f = Cn \ 1 \ r\text{-singleton-encode } [r\text{-not}]$

have $\text{eval } r\text{-badblock } [0, v] = \text{eval } ?f \ [v]$

unfolding *r-badblock-def* **by** *simp*

also have $\dots = \text{eval } r\text{-singleton-encode } [\text{the } (\text{eval } r\text{-not } [v])]$

by *simp*

also have $\dots \Downarrow = \text{list-encode } [1 - v]$

by *simp*

finally show *?case* **by** *simp*

next

case (*Suc n*)

let $?g = Cn \ 3 \ r\text{-cons } [r\text{-constn } 2 \ 0, Id \ 3 \ 1]$

have $\text{recfn } 3 \ ?g$ **by** *simp*

have $\text{eval } r\text{-badblock } [(Suc \ n), v] = \text{eval } ?g \ [n, \text{the } (\text{eval } r\text{-badblock } [n, v]), v]$

using $\langle \text{recfn } 3 \ ?g \rangle \ Suc$ **by** (*simp add: r-badblock-def*)

also have $\dots = \text{eval } ?g \ [n, \text{list-encode } (\text{replicate } n \ 0 \ @ \ [1 - v]), v]$

using *Suc* **by** *simp*

also have $\dots = \text{eval } r\text{-cons } [0, \text{list-encode } (\text{replicate } n \ 0 \ @ \ [1 - v])]$

by *simp*

also have $\dots \Downarrow = e\text{-cons } 0 \ (\text{list-encode } (\text{replicate } n \ 0 \ @ \ [1 - v]))$

by *simp*

also have $\dots \Downarrow = \text{list-encode } (0 \ \# \ (\text{replicate } n \ 0 \ @ \ [1 - v]))$

by *simp*

also have $\dots \Downarrow = \text{list-encode } (\text{replicate } (Suc \ n) \ 0 \ @ \ [1 - v])$

by *simp*

finally show *?case* **by** *simp*

qed

lemma *r-badblock-only-01*: $e\text{-nth } (\text{the } (\text{eval } r\text{-badblock } [n, v])) \ i \leq 1$

using *r-badblock* **by** (*simp add: nth-append*)

lemma *r-badblock-last*: $e\text{-nth } (\text{the } (\text{eval } r\text{-badblock } [n, v])) \ n = 1 - v$

using *r-badblock* **by** (*simp add: nth-append*)

The following function computes the next prefix from the current one. In other words, it performs Steps 1 and 2.

definition *r-next* \equiv

Cn 1 r-append

[Id 1 0,

Cn 1 r-badblock

[Cn 1 r-sub [r-search, r-length],

Cn 1 r-phi [r-s, r-search]]]

lemma *r-next-recfn* [*simp*]: $\text{recfn } 1 \ r\text{-next}$

unfolding *r-next-def* **by** *simp*

The name *next* is unavailable, so we go for *nxt*.

abbreviation *nxt* $:: \text{partial1}$ **where**

$\text{nxt } b \equiv \text{eval } r\text{-next } [b]$

lemma *nxt-diverg*:

assumes $\text{search } b \uparrow$

shows $\text{nxt } b \uparrow$

unfolding *r-next-def* **using** *assms* **by** (*simp add: Let-def*)

lemma *nxt-converg*:

assumes *search b* $\downarrow = y$

shows *nxt b* $\downarrow =$

e-append b (list-encode (replicate (y - e-length b) 0 @ [1 - the (φ (the (s b)) y)]))

unfolding *r-next-def* **using** *assms r-badblock search-converg phi-def eval-rs*

by *fastforce*

lemma *nxt-search-diverg*:

assumes *nxt b* \uparrow

shows *search b* \uparrow

proof (*rule ccontr*)

assume *search b* \downarrow

then obtain *y* **where** *search b* $\downarrow = y$ **by** *auto*

then show *False*

using *nxt-converg assms* **by** *simp*

qed

If Step 1 finds a *y*, the hypothesis *S(b)* is incorrect for the new prefix.

lemma *nxt-wrong-hyp*:

assumes *nxt b* $\downarrow = b'$ **and** *s b* $\downarrow = i$

shows $\exists y < e\text{-length } b'. \varphi i y \downarrow \neq e\text{-nth } b' y$

proof –

obtain *y* **where** *y: search b* $\downarrow = y$

using *assms nxt-diverg* **by** *fastforce*

then have *y-len: y* $\geq e\text{-length } b$

using *assms search-converg(2)* **by** *fastforce*

then have *b': b' =*

(e-append b (list-encode (replicate (y - e-length b) 0 @ [1 - the ($\varphi i y$)]))

using *y assms nxt-converg* **by** *simp*

then have *e-nth b' y = 1 - the ($\varphi i y$)*

using *y-len e-nth-append-big r-badblock r-badblock-last* **by** *auto*

moreover have $\varphi i y \downarrow$

using *search-converg y y-len assms(2)* **by** *fastforce*

ultimately have $\varphi i y \downarrow \neq e\text{-nth } b' y$

by (*metis gr-zeroI less-numeral-extra(4) less-one option.sel zero-less-diff*)

moreover have *e-length b' = Suc y*

using *y-len e-length-append b'* **by** *auto*

ultimately show *?thesis* **by** *auto*

qed

If Step 1 diverges, the hypothesis *S(b)* refers to a non-total function.

lemma *nxt-nontotal-hyp*:

assumes *nxt b* \uparrow **and** *s b* $\downarrow = i$

shows $\exists x. \varphi i x \uparrow$

using *nxt-search-diverg[OF assms(1)] search-diverg[OF assms(2)]* **by** *auto*

The process only ever extends the given prefix.

lemma *nxt-stable*:

assumes *nxt b* $\downarrow = b'$

shows $\forall x < e\text{-length } b. e\text{-nth } b x = e\text{-nth } b' x$

proof –

obtain *y* **where** *y: search b* $\downarrow = y$

using *assms nxt-diverg* **by** *fastforce*

```

then have  $y \geq e\text{-length } b$ 
  using search-converg(2) eval-rs rs-total by fastforce
show ?thesis
proof (rule allI, rule impI)
  fix  $x$  assume  $x < e\text{-length } b$ 
  let  $?i = \text{the } (s \ b)$ 
  have  $b': b' =$ 
    (e-append  $b$  (list-encode (replicate ( $y - e\text{-length } b$ )  $0$  @ [ $1 - \text{the } (\varphi \ ?i \ y)$ ]))))
  using assms next-converg[OF y] by auto
  then show  $e\text{-nth } b \ x = e\text{-nth } b' \ x$ 
    using e-nth-append-small  $\langle x < e\text{-length } b \rangle$  by auto
qed
qed

```

The following properties of *r-next* will be used to show that some of the constructed functions are in the class *V*.

```

lemma next-append-01:
  assumes next  $b \downarrow = b'$ 
  shows  $\forall x. x \geq e\text{-length } b \wedge x < e\text{-length } b' \longrightarrow e\text{-nth } b' \ x = 0 \vee e\text{-nth } b' \ x = 1$ 
proof -
  obtain  $y$  where  $y: \text{search } b \downarrow = y$ 
    using assms next-diverg by fastforce
  let  $?i = \text{the } (s \ b)$ 
  have  $b': b' = (\text{e-append } b \ (\text{list-encode } (\text{replicate } (y - e\text{-length } b) \ 0 \ @ \ [1 - \text{the } (\varphi \ ?i \ y)])))$ 
    (is  $b' = (\text{e-append } b \ ?z)$ )
    using assms y next-converg prod-encode-eq by auto
  show ?thesis
proof (rule allI, rule impI)
  fix  $x$  assume  $x: e\text{-length } b \leq x \wedge x < e\text{-length } b'$ 
  then have  $e\text{-nth } b' \ x = e\text{-nth } ?z \ (x - e\text{-length } b)$ 
    using b' e-nth-append-big by blast
  then show  $e\text{-nth } b' \ x = 0 \vee e\text{-nth } b' \ x = 1$ 
    by (metis less-one nat-less-le option.sel r-badblock r-badblock-only-01)
qed
qed

```

```

lemma next-monotone:
  assumes next  $b \downarrow = b'$ 
  shows  $e\text{-length } b < e\text{-length } b'$ 
proof -
  obtain  $y$  where  $y: \text{search } b \downarrow = y$ 
    using assms next-diverg by fastforce
  let  $?i = \text{the } (s \ b)$ 
  have  $b': b' =$ 
    (e-append  $b$  (list-encode (replicate ( $y - e\text{-length } b$ )  $0$  @ [ $1 - \text{the } (\varphi \ ?i \ y)$ ]))))
  using assms y next-converg prod-encode-eq by auto
  then show ?thesis using e-length-append by auto
qed

```

The next function computes the prefixes after each iteration of the process *r-next* when started with the list $[j]$.

```

definition r-prefixes :: recf where
  r-prefixes  $\equiv Pr \ 1 \ r\text{-singleton-encode } (Cn \ 3 \ r\text{-next } [Id \ 3 \ 1])$ 

```

```

lemma r-prefixes-recfn [simp]: recfn  $2 \ r\text{-prefixes}$ 

```

unfolding *r-prefixes-def* **by** (*simp add: Let-def*)

abbreviation *prefixes* :: *partial2* **where**
prefixes *t j* \equiv *eval r-prefixes* [*t*, *j*]

lemma *prefixes-at-0*: *prefixes* 0 *j* $\downarrow =$ *list-encode* [*j*]
unfolding *r-prefixes-def* **by** *simp*

lemma *prefixes-at-Suc*:
assumes *prefixes* *t j* \downarrow (**is** *?b* \downarrow)
shows *prefixes* (*Suc t*) *j* = *next* (*the ?b*)
using *r-prefixes-def* *assms* **by** *auto*

lemma *prefixes-at-Suc'*:
assumes *prefixes* *t j* $\downarrow =$ *b*
shows *prefixes* (*Suc t*) *j* = *next* *b*
using *r-prefixes-def* *assms* **by** *auto*

lemma *prefixes-prod-encode*:
assumes *prefixes* *t j* \downarrow
obtains *b* **where** *prefixes* *t j* $\downarrow =$ *b*
using *assms surj-prod-encode* **by** *force*

lemma *prefixes-converg-le*:
assumes *prefixes* *t j* \downarrow **and** *t'* \leq *t*
shows *prefixes* *t' j* \downarrow
using *r-prefixes-def* *assms eval-Pr-converg-le*[*of 1 - - [j]*]
by *simp*

lemma *prefixes-diverg-add*:
assumes *prefixes* *t j* \uparrow
shows *prefixes* (*t + d*) *j* \uparrow
using *r-prefixes-def* *assms eval-Pr-diverg-add*[*of 1 - - [j]*]
by *simp*

Many properties of *r-prefixes* can be derived from similar properties of *r-next*.

lemma *prefixes-length*:
assumes *prefixes* *t j* $\downarrow =$ *b*
shows *e-length* *b* $>$ *t*
proof (*insert assms, induction t arbitrary: b*)
case 0
then show *?case* **using** *prefixes-at-0 prod-encode-eq* **by** *auto*
next
case (*Suc t*)
then have *prefixes* *t j* \downarrow
using *prefixes-converg-le Suc-n-not-le-n nat-le-linear* **by** *blast*
then obtain *b'* **where** *b'*: *prefixes* *t j* $\downarrow =$ *b'*
using *prefixes-prod-encode* **by** *blast*
with *Suc* **have** *e-length* *b'* $>$ *t* **by** *simp*
have *prefixes* (*Suc t*) *j* = *next* *b'*
using *b' prefixes-at-Suc'* **by** *simp*
with *Suc* **have** *next* *b'* $\downarrow =$ *b* **by** *simp*
then have *e-length* *b'* $<$ *e-length* *b*
using *next-monotone* **by** *simp*
then show *?case* **using** \langle *e-length* *b'* $>$ *t* \rangle **by** *simp*
qed

lemma *prefixes-monotone*:

assumes *prefixes* $t\ j \downarrow = b$ and *prefixes* $(t + d)\ j \downarrow = b'$
 shows $e\text{-length}\ b \leq e\text{-length}\ b'$

proof (insert *assms*, induction d arbitrary: b')

case 0

then show ?case using *prod-encode-eq* by *simp*

next

case (*Suc* d)

moreover have $t + d \leq t + \text{Suc}\ d$ by *simp*

ultimately have *prefixes* $(t + d)\ j \downarrow$

using *prefixes-converg-le* by *blast*

then obtain b'' where b'' : *prefixes* $(t + d)\ j \downarrow = b''$

using *prefixes-prod-encode* by *blast*

with *Suc* have *prefixes* $(t + \text{Suc}\ d)\ j = \text{next}\ b''$

by (*simp* add: *prefixes-at-Suc'*)

with *Suc* have $\text{next}\ b'' \downarrow = b'$ by *simp*

then show ?case using *next-monotone* *Suc* b'' by *fastforce*

qed

lemma *prefixes-stable*:

assumes *prefixes* $t\ j \downarrow = b$ and *prefixes* $(t + d)\ j \downarrow = b'$

shows $\forall x < e\text{-length}\ b. e\text{-nth}\ b\ x = e\text{-nth}\ b'\ x$

proof (insert *assms*, induction d arbitrary: b')

case 0

then show ?case using *prod-encode-eq* by *simp*

next

case (*Suc* d)

moreover have $t + d \leq t + \text{Suc}\ d$ by *simp*

ultimately have *prefixes* $(t + d)\ j \downarrow$

using *prefixes-converg-le* by *blast*

then obtain b'' where b'' : *prefixes* $(t + d)\ j \downarrow = b''$

using *prefixes-prod-encode* by *blast*

with *Suc* have *prefixes* $(t + \text{Suc}\ d)\ j = \text{next}\ b''$

by (*simp* add: *prefixes-at-Suc'*)

with *Suc* have b' : $\text{next}\ b'' \downarrow = b'$ by *simp*

show $\forall x < e\text{-length}\ b. e\text{-nth}\ b\ x = e\text{-nth}\ b'\ x$

proof (rule *allI*, rule *impI*)

fix x assume $x < e\text{-length}\ b$

then have $e\text{-nth}\ b\ x = e\text{-nth}\ b''\ x$

using *Suc* b'' by *simp*

moreover have $x \leq e\text{-length}\ b''$

using x *prefixes-monotone* b'' *Suc* by *fastforce*

ultimately show $e\text{-nth}\ b\ x = e\text{-nth}\ b'\ x$

using b'' *next-stable* *Suc* b' *prefixes-monotone* x

by (*metis* *leD* *le-neq-implies-less*)

qed

qed

lemma *prefixes-tl-only-01*:

assumes *prefixes* $t\ j \downarrow = b$

shows $\forall x > 0. e\text{-nth}\ b\ x = 0 \vee e\text{-nth}\ b\ x = 1$

proof (insert *assms*, induction t arbitrary: b)

case 0

then show ?case using *prefixes-at-0* *prod-encode-eq* by *auto*

next

```

case (Suc t)
then have prefixes t j ↓
  using prefixes-converg-le Suc-n-not-le-n nat-le-linear by blast
then obtain b' where b': prefixes t j ↓= b'
  using prefixes-prod-encode by blast
show  $\forall x > 0. e\text{-nth } b \ x = 0 \vee e\text{-nth } b \ x = 1$ 
proof (rule allI, rule impI)
  fix x :: nat
  assume x: x > 0
  show e-nth b x = 0  $\vee$  e-nth b x = 1
  proof (cases x < e-length b')
    case True
    then show ?thesis
      using Suc b' prefixes-at-Suc' nst-stable x by metis
  next
    case False
    then show ?thesis
      using Suc.prem b' prefixes-at-Suc' nst-append-01 by auto
  qed
qed
qed

```

```

lemma prefixes-hd:
  assumes prefixes t j ↓= b
  shows e-nth b 0 = j
proof -
  obtain b' where b': prefixes 0 j ↓= b'
    by (simp add: prefixes-at-0)
  then have b' = list-encode [j]
    by (simp add: prod-encode-eq prefixes-at-0)
  then have e-nth b' 0 = j by simp
  then show e-nth b 0 = j
    using assms prefixes-stable[OF b', of t b] prefixes-length[OF b'] by simp
qed

```

```

lemma prefixes-nontotal-hyp:
  assumes prefixes t j ↓= b
  and prefixes (Suc t) j ↑
  and s b ↓= i
  shows  $\exists x. \varphi \ i \ x \ \uparrow$ 
  using nst-nontotal-hyp[OF - assms(3)] assms(2) prefixes-at-Suc'[OF assms(1)] by simp

```

We now consider the two cases from the proof sketch.

abbreviation *case-two* j $\equiv \exists t. \text{prefixes } t \ j \ \uparrow$

abbreviation *case-one* j $\equiv \neg \text{case-two } j$

In Case 2 there is a maximum convergent iteration because iteration 0 converges.

```

lemma case-two:
  assumes case-two j
  shows  $\exists t. (\forall t' \leq t. \text{prefixes } t' \ j \ \downarrow) \wedge (\forall t' > t. \text{prefixes } t' \ j \ \uparrow)$ 
proof -
  let ?P =  $\lambda t. \text{prefixes } t \ j \ \uparrow$ 
  define t0 where t0 = Least ?P
  then have ?P t0
    using assms LeastI-ex[of ?P] by simp

```

then have *diverg*: $?P\ t$ **if** $t \geq t_0$ **for** t
using *prefixes-converg-le* **that by** *blast*
from t_0 -*def* **have** *converg*: $\neg ?P\ t$ **if** $t < t_0$ **for** t
using *Least-le*[of $?P$] **that not-less by** *blast*
have $t_0 > 0$
proof (*rule ccontr*)
assume $\neg 0 < t_0$
then have $t_0 = 0$ **by** *simp*
with $\langle ?P\ t_0 \rangle$ *prefixes-at-0* **show** *False* **by** *simp*
qed
let $?t = t_0 - 1$
have $\forall t' \leq ?t.$ *prefixes* $t'\ j \downarrow$
using *converg* $\langle 0 < t_0 \rangle$ **by** *auto*
moreover have $\forall t' > ?t.$ *prefixes* $t'\ j \uparrow$
using *diverg* **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

Having completed the modelling of the process, we can now define the functions ψ_j it computes. The value $\psi_j(x)$ is computed by running *r-prefixes* until the prefix is longer than x and then taking the x -th element of the prefix.

definition *r-psi* \equiv
let $f = Cn\ 3\ r-less\ [Id\ 3\ 2,\ Cn\ 3\ r-length\ [Cn\ 3\ r-prefixes\ [Id\ 3\ 0,\ Id\ 3\ 1]]]$
in $Cn\ 2\ r-nth\ [Cn\ 2\ r-prefixes\ [Mn\ 2\ f,\ Id\ 2\ 0],\ Id\ 2\ 1]$

lemma *r-psi-recfn*: *recfn* $2\ r-psi$
unfolding *r-psi-def* **by** *simp*

abbreviation *psi* $::$ *partial2* ($\langle \psi \rangle$) **where**
 $\psi\ j\ x \equiv eval\ r-psi\ [j,\ x]$

lemma *psi-in-P2*: $\psi \in \mathcal{P}^2$
using *r-psi-recfn* **by** *auto*

The values of ψ can be read off the prefixes.

lemma *psi-eq-nth-prefix*:
assumes *prefixes* $t\ j \downarrow = b$ **and** *e-length* $b > x$
shows $\psi\ j\ x \downarrow = e-nth\ b\ x$
proof –
let $?f = Cn\ 3\ r-less\ [Id\ 3\ 2,\ Cn\ 3\ r-length\ [Cn\ 3\ r-prefixes\ [Id\ 3\ 0,\ Id\ 3\ 1]]]$
let $?P = \lambda t.$ *prefixes* $t\ j \downarrow \wedge e-length\ (the\ (prefixes\ t\ j)) > x$
from *assms* **have** *ex-t*: $\exists t.$ $?P\ t$ **by** *auto*
define t_0 **where** $t_0 = Least\ ?P$
then have $?P\ t_0$
using *LeastI-ex*[*OF ex-t*] **by** *simp*
from *ex-t* **have** *not-P*: $\neg ?P\ t$ **if** $t < t_0$ **for** t
using *ex-t* **that** *Least-le*[of $?P$] *not-le* t_0 -*def* **by** *auto*

have $?P\ t$ **using** *assms* **by** *simp*
with *not-P* **have** $t_0 \leq t$ **using** *leI* **by** *blast*
then obtain b_0 **where** b_0 : *prefixes* $t_0\ j \downarrow = b_0$
using *assms*(1) *prefixes-converg-le* **by** *blast*

have *eval* $?f\ [t_0,\ j,\ x] \downarrow = 0$
proof –

have *eval* (*Cn 3 r-prefixes* [*Id 3 0*, *Id 3 1*]) [*t*₀, *j*, *x*] $\downarrow = b_0$
using *b0* **by** *simp*
then show *?thesis* **using** $\langle ?P t_0 \rangle$ **by** *simp*
qed
moreover have *eval* *?f* [*t*, *j*, *x*] $\downarrow \neq 0$ **if** *t* < *t*₀ **for** *t*
proof –
obtain *bt* **where** *bt*: *prefixes t j* $\downarrow = bt$
using *prefixes-converg-le*[*of t_0 j t*] *b0* $\langle t < t_0 \rangle$ **by** *auto*
moreover have $\neg ?P t$
using *that not-P* **by** *simp*
ultimately have *e-length bt* $\leq x$ **by** *simp*
moreover have *eval* (*Cn 3 r-prefixes* [*Id 3 0*, *Id 3 1*]) [*t*, *j*, *x*] $\downarrow = bt$
using *bt* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
ultimately have *eval* (*Mn 2 ?f*) [*j*, *x*] $\downarrow = t_0$
using *eval-Mn-convergI*[*of 2 ?f [j, x] t_0*] **by** *simp*
then have $\psi j x \downarrow = e\text{-nth } b_0 x$
unfolding *r-psi-def* **using** *b0* **by** *simp*
then show *?thesis*
using $\langle t_0 \leq t \rangle$ *assms(1)* *prefixes-stable*[*of t_0 j b_0 t - t_0 b*] *b0* $\langle ?P t_0 \rangle$
by *simp*
qed
lemma *psi-converg-imp-prefix*:
assumes $\psi j x \downarrow$
shows $\exists t b. \text{prefixes } t j \downarrow = b \wedge e\text{-length } b > x$
proof –
let *?f* = *Cn 3 r-less* [*Id 3 2*, *Cn 3 r-length* [*Cn 3 r-prefixes* [*Id 3 0*, *Id 3 1*]]]
have *eval* (*Mn 2 ?f*) [*j*, *x*] \downarrow
proof (*rule ccontr*)
assume $\neg \text{eval } (Mn 2 ?f) [j, x] \downarrow$
then have *eval* (*Mn 2 ?f*) [*j*, *x*] \uparrow **by** *simp*
then have $\psi j x \uparrow$
unfolding *r-psi-def* **by** *simp*
then show *False*
using *assms* **by** *simp*
qed
then obtain *t* **where** *t*: *eval* (*Mn 2 ?f*) [*j*, *x*] $\downarrow = t$
by *blast*
have *recfn 2* (*Mn 2 ?f*) **by** *simp*
then have *f-zero*: *eval* *?f* [*t*, *j*, *x*] $\downarrow = 0$
using *eval-Mn-convergE*[*OF - t*]
by (*metis* (*no-types*, *lifting*) *One-nat-def* *Suc-1 length-Cons list.size(3)*)
have *prefixes t j* \downarrow
proof (*rule ccontr*)
assume $\neg \text{prefixes } t j \downarrow$
then have *prefixes t j* \uparrow **by** *simp*
then have *eval* *?f* [*t*, *j*, *x*] \uparrow **by** *simp*
with *f-zero* **show** *False* **by** *simp*
qed
then obtain *b'* **where** *b'*: *prefixes t j* $\downarrow = b'$ **by** *auto*
moreover have *e-length b'* > *x*
proof (*rule ccontr*)
assume $\neg e\text{-length } b' > x$
then have *eval* *?f* [*t*, *j*, *x*] $\downarrow = 1$

using b' **by** *simp*
with f -zero **show** *False* **by** *simp*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *psi-converg-imp-prefix'*:
assumes $\psi j x \downarrow$
shows $\exists t b. \text{prefixes } t j \downarrow = b \wedge e\text{-length } b > x \wedge \psi j x \downarrow = e\text{-nth } b x$
using *psi-converg-imp-prefix[OF assms]* *psi-eq-nth-prefix* **by** *blast*

In both Case 1 and 2, ψ_j starts with j .

lemma *psi-at-0*: $\psi j 0 \downarrow = j$
using *prefixes-hd prefixes-length psi-eq-nth-prefix prefixes-at-0* **by** *fastforce*

In Case 1, ψ_j is total and made up of j followed by zeros and ones, just as required by the definition of V_1 .

lemma *case-one-psi-total*:
assumes *case-one* j **and** $x > 0$
shows $\psi j x \downarrow = 0 \vee \psi j x \downarrow = 1$
proof –
obtain b **where** $b: \text{prefixes } x j \downarrow = b$
using *assms(1)* **by** *auto*
then have $e\text{-length } b > x$
using *prefixes-length* **by** *simp*
then have $\psi j x \downarrow = e\text{-nth } b x$
using b *psi-eq-nth-prefix* **by** *simp*
moreover have $e\text{-nth } b x = 0 \vee e\text{-nth } b x = 1$
using *prefixes-tl-only-01[OF b] assms(2)* **by** *simp*
ultimately show $\psi j x \downarrow = 0 \vee \psi j x \downarrow = 1$
by *simp*
qed

In Case 2, ψ_j is defined only for a prefix starting with j and continuing with zeros and ones. This prefix corresponds to ja from the definition of V_2 .

lemma *case-two-psi-only-prefix*:
assumes *case-two* j
shows $\exists y. (\forall x. 0 < x \wedge x < y \longrightarrow \psi j x \downarrow = 0 \vee \psi j x \downarrow = 1) \wedge$
 $(\forall x \geq y. \psi j x \uparrow)$

proof –
obtain t **where**
 $t\text{-le}: \forall t' \leq t. \text{prefixes } t' j \downarrow$ **and**
 $t\text{-gr}: \forall t' > t. \text{prefixes } t' j \uparrow$
using *assms case-two* **by** *blast*
then obtain b **where** $b: \text{prefixes } t j \downarrow = b$
by *auto*
let $?y = e\text{-length } b$
have $\psi j x \downarrow = 0 \vee \psi j x \downarrow = 1$ **if** $x > 0 \wedge x < ?y$ **for** x
using $t\text{-le } b$ **that** **by** (*metis prefixes-tl-only-01 psi-eq-nth-prefix*)
moreover have $\psi j x \uparrow$ **if** $x \geq ?y$ **for** x
proof (*rule ccontr*)
assume $\psi j x \downarrow$
then obtain $t' b'$ **where** $t': \text{prefixes } t' j \downarrow = b'$ **and** $e\text{-length } b' > x$
using *psi-converg-imp-prefix* **by** *blast*
then have $e\text{-length } b' > ?y$

```

    using that by simp
  with t' have t' > t
    using prefixes-monotone b by (metis add-diff-inverse-nat leD)
  with t' t-gr show False by simp
qed
ultimately show ?thesis by auto
qed

definition longest-prefix :: nat ⇒ nat where
  longest-prefix j ≡ THE y. (∀ x < y. ψ j x ↓) ∧ (∀ x ≥ y. ψ j x ↑)

lemma longest-prefix:
  assumes case-two j and z = longest-prefix j
  shows (∀ x < z. ψ j x ↓) ∧ (∀ x ≥ z. ψ j x ↑)
proof -
  let ?P = λz. (∀ x < z. ψ j x ↓) ∧ (∀ x ≥ z. ψ j x ↑)
  obtain y where y:
    ∀ x. 0 < x ∧ x < y ⟶ ψ j x ↓ = 0 ∨ ψ j x ↓ = 1
    ∀ x ≥ y. ψ j x ↑
  using case-two-psi-only-prefix[OF assms(1)] by auto
  have ?P (THE z. ?P z)
  proof (rule theI[of ?P y])
    show ?P y
    proof
      show ∀ x < y. ψ j x ↓
      proof (rule allI, rule impI)
        fix x assume x < y
        show ψ j x ↓
        proof (cases x = 0)
          case True
            then show ?thesis using psi-at-0 by simp
          next
            case False
              then show ?thesis using y(1) ⟨x < y⟩ by auto
        qed
      show ∀ x ≥ y. ψ j x ↑ using y(2) by simp
    qed
  show z = y if ?P z for z
  proof (rule ccontr, cases z < y)
    case True
      moreover assume z ≠ y
      ultimately show False
      using that ⟨?P y⟩ by auto
    next
      case False
        moreover assume z ≠ y
        then show False
        using that ⟨?P y⟩ y(2) by (meson linorder-cases order-refl)
    qed
  qed
  then have (∀ x < (THE z. ?P z). ψ j x ↓) ∧ (∀ x ≥ (THE z. ?P z). ψ j x ↑)
  by blast
  moreover have longest-prefix j = (THE z. ?P z)
  unfolding longest-prefix-def by simp
  ultimately show ?thesis using assms(2) by metis

```

qed

lemma *case-two-psi-longest-prefix*:

assumes *case-two j* **and** *y = longest-prefix j*
shows $(\forall x. 0 < x \wedge x < y \longrightarrow \psi j x \downarrow = 0 \vee \psi j x \downarrow = 1) \wedge$
 $(\forall x \geq y. \psi j x \uparrow)$
using *assms longest-prefix case-two-psi-only-prefix*
by (*metis prefixes-tl-only-01 psi-converg-imp-prefix'*)

The prefix cannot be empty because the process starts with prefix $[j]$.

lemma *longest-prefix-gr-0*:

assumes *case-two j*
shows *longest-prefix j > 0*
using *assms case-two-psi-longest-prefix psi-at-0* **by** *force*

lemma *psi-not-divergent-init*:

assumes *prefixes t j \downarrow = b*
shows $(\psi j) \triangleright (e\text{-length } b - 1) = b$
proof (*intro initI*)
show $0 < e\text{-length } b$
using *assms prefixes-length* **by** *fastforce*
show $\psi j x \downarrow = e\text{-nth } b x$ **if** $x < e\text{-length } b$ **for** x
using *that assms psi-eq-nth-prefix* **by** *simp*

qed

In Case 2, the strategy S outputs a non-total hypothesis on some prefix of ψ_j .

lemma *case-two-nontotal-hyp*:

assumes *case-two j*
shows $\exists n < \text{longest-prefix } j. \neg \text{total1 } (\varphi (\text{the } (s ((\psi j) \triangleright n))))$
proof –
obtain t **where** $\forall t' \leq t. \text{prefixes } t' j \downarrow$ **and** $t\text{-gr}: \forall t' > t. \text{prefixes } t' j \uparrow$
using *assms case-two* **by** *blast*
then obtain b **where** $b: \text{prefixes } t j \downarrow = b$
by *auto*
moreover obtain i **where** $i: s b \downarrow = i$
using *eval-rs* **by** *fastforce*
moreover have $\text{div}: \text{prefixes } (\text{Suc } t) j \uparrow$
using $t\text{-gr}$ **by** *simp*
ultimately have $\exists x. \varphi i x \uparrow$
using *prefixes-nontotal-hyp* **by** *simp*
then obtain x **where** $\varphi i x \uparrow$ **by** *auto*
moreover have $\text{init}: \psi j \triangleright (e\text{-length } b - 1) = b$ (**is** $- \triangleright ?n = b$)
using *psi-not-divergent-init[OF b]* **by** *simp*
ultimately have $\varphi (\text{the } (s (\psi j \triangleright ?n))) x \uparrow$
using i **by** *simp*
then have $\neg \text{total1 } (\varphi (\text{the } (s (\psi j \triangleright ?n))))$
by *auto*
moreover have $?n < \text{longest-prefix } j$
using *case-two-psi-longest-prefix init b div psi-eq-nth-prefix*
by (*metis length-init lessI not-le-imp-less option.simps(3)*)
ultimately show *?thesis* **by** *auto*

qed

Consequently, in Case 2 the strategy does not TOTAL-learn any function starting with the longest prefix of ψ_j .

lemma *case-two-not-learn*:

assumes *case-two* j
and $f \in \mathcal{R}$
and $\bigwedge x. x < \text{longest-prefix } j \implies f x = \psi j x$
shows $\neg \text{learn-total } \varphi \{f\} s$

proof –

obtain n **where** n :
 $n < \text{longest-prefix } j$
 $\neg \text{total1 } (\varphi (\text{the } (s (\psi j \triangleright n))))$
using *case-two-nontotal-hyp*[*OF assms*(1)] **by** *auto*
have $f \triangleright n = \psi j \triangleright n$
using *assms*(3) n (1) **by** (*intro init-eqI*) *auto*
with n (2) **show** *?thesis* **by** (*metis R1-imp-total1 learn-totalE*(3) *singletonI*)

qed

In Case 1 the strategy outputs a wrong hypothesis on infinitely many prefixes of ψ_j and thus does not learn ψ_j in the limit, much less in the sense of TOTAL.

lemma *case-one-wrong-hyp*:

assumes *case-one* j
shows $\exists n > k. \varphi (\text{the } (s ((\psi j) \triangleright n))) \neq \psi j$

proof –

have *all-t*: $\forall t. \text{prefixes } t j \downarrow$
using *assms* **by** *simp*
then obtain b **where** $b: \text{prefixes } (\text{Suc } k) j \downarrow = b$
by *auto*
then have *length*: $e\text{-length } b > \text{Suc } k$
using *prefixes-length* **by** *simp*
then have *init*: $\psi j \triangleright (e\text{-length } b - 1) = b$
using *psi-not-divergent-init* b **by** *simp*
obtain i **where** $i: s b \downarrow = i$
using *eval-rs* **by** *fastforce*
from *all-t* **obtain** b' **where** $b': \text{prefixes } (\text{Suc } (\text{Suc } k)) j \downarrow = b'$
by *auto*
then have $\psi j \triangleright (e\text{-length } b' - 1) = b'$
using *psi-not-divergent-init* **by** *simp*
moreover have $\exists y < e\text{-length } b'. \varphi i y \downarrow \neq e\text{-nth } b' y$
using *next-wrong-hyp* $b b' i$ *prefixes-at-Suc* **by** *auto*
ultimately have $\exists y < e\text{-length } b'. \varphi i y \neq \psi j y$
using b' *psi-eq-nth-prefix* **by** *auto*
then have $\varphi i \neq \psi j$ **by** *auto*
then show *?thesis*
using *init length i* **by** (*metis Suc-less-eq length-init option.sel*)

qed

lemma *case-one-not-learn*:

assumes *case-one* j
shows $\neg \text{learn-lim } \varphi \{\psi j\} s$

proof (*rule infinite-hyp-wrong-not-Lim*[of ψj])

show $\psi j \in \{\psi j\}$ **by** *simp*
show $\forall n. \exists m > n. \varphi (\text{the } (s (\psi j \triangleright m))) \neq \psi j$
using *case-one-wrong-hyp*[*OF assms*] **by** *simp*

qed

lemma *case-one-not-learn-V*:

assumes *case-one* j **and** $j \geq 2$ **and** $\varphi j = \psi j$
shows $\neg \text{learn-lim } \varphi V\text{-constotal } s$

```

proof –
  have  $\psi j \in V\text{-constotal-1}$ 
  proof –
    define  $p$  where  $p = (\lambda x. (\psi j) (x + 1))$ 
    have  $p \in \mathcal{R}_{01}$ 
    proof –
      from  $p\text{-def}$  have  $p \in \mathcal{P}$ 
      using  $\text{skip-P1}[of \ \psi j \ 1] \ \text{psi-in-P2} \ \text{P2-proj-P1}$  by  $\text{blast}$ 
      moreover have  $p \ x \ \downarrow = 0 \ \vee \ p \ x \ \downarrow = 1$  for  $x$ 
      using  $p\text{-def} \ \text{assms}(1) \ \text{case-one-psi-total}$  by  $\text{auto}$ 
      moreover from  $\text{this}$  have  $\text{total1 } p$  by  $\text{fast}$ 
      ultimately show  $?thesis$  using  $R\text{Pred1-def}$  by  $\text{auto}$ 
    qed
    moreover have  $\psi j = [j] \odot p$ 
    by  $(\text{intro } \text{prepend-eqI}, \ \text{simp } \text{add: } \text{psi-at-0}, \ \text{simp } \text{add: } p\text{-def})$ 
    ultimately show  $?thesis$  using  $\text{assms}(2,3) \ V\text{-constotal-1-def}$  by  $\text{blast}$ 
  qed
  then have  $\psi j \in V\text{-constotal}$  using  $V\text{-constotal-def}$  by  $\text{auto}$ 
  moreover have  $\neg \text{learn-lim } \varphi \ \{\psi j\} \ s$ 
  using  $\text{case-one-not-learn } \text{assms}(1)$  by  $\text{simp}$ 
  ultimately show  $?thesis$  using  $\text{learn-lim-closed-subseteq}$  by  $\text{auto}$ 
qed

```

The next lemma embodies the construction of χ followed by the application of Kleene's fixed-point theorem as described in the proof sketch.

```

lemma  $\text{goedel-after-prefixes}$ :
  fixes  $vs :: \text{nat list}$  and  $m :: \text{nat}$ 
  shows  $\exists n \geq m. \ \varphi \ n = vs \ @ \ [n] \odot 0^\infty$ 
proof –
  define  $f :: \text{partial1}$  where  $f \equiv vs \odot 0^\infty$ 
  then have  $f \in \mathcal{R}$ 
  using  $\text{almost0-in-R1}$  by  $\text{auto}$ 
  then obtain  $n$  where  $n$ :
     $n \geq m$ 
     $\varphi \ n = (\lambda x. \ \text{if } x = \text{length } vs \ \text{then } \text{Some } n \ \text{else } f \ x)$ 
    using  $\text{goedel-at}[of \ f \ m \ \text{length } vs]$  by  $\text{auto}$ 
  moreover have  $\varphi \ n \ x = (vs \ @ \ [n] \odot 0^\infty) \ x$  for  $x$ 
proof –
  consider  $x < \text{length } vs \ | \ x = \text{length } vs \ | \ x > \text{length } vs$ 
  by  $\text{linarith}$ 
  then show  $?thesis$ 
  using  $n \ f\text{-def}$  by  $(\text{cases}) \ (\text{auto } \text{simp } \text{add: } \text{prepend-associative})$ 
qed
  ultimately show  $?thesis$  by  $\text{blast}$ 
qed

```

If Case 2 holds for a $j \geq 2$ with $\varphi_j = \psi_j$, that is, if $\psi_j \in V_1$, then there is a function in V , namely ψ_j , on which S fails. Therefore S does not learn V .

```

lemma  $\text{case-two-not-learn-V}$ :
  assumes  $\text{case-two } j$  and  $j \geq 2$  and  $\varphi \ j = \psi \ j$ 
  shows  $\neg \text{learn-total } \varphi \ V\text{-constotal } s$ 
proof –
  define  $z$  where  $z = \text{longest-prefix } j$ 
  then have  $z > 0$ 
  using  $\text{longest-prefix-gr-0}[OF \ \text{assms}(1)]$  by  $\text{simp}$ 

```

```

define vs where vs = prefix ( $\psi$  j) (z - 1)
then have vs ! 0 = j
  using psi-at-0  $\langle z > 0 \rangle$  by simp
define a where a = tl vs
then have vs: vs = j # a
  using vs-def  $\langle vs ! 0 = j \rangle$ 
  by (metis length-Suc-conv length-prefix list.sel(3) nth-Cons-0)
obtain k where k: k ≥ 2 and phi-k:  $\varphi$  k = j # a @ [k] ⊙ 0∞
  using goedel-after-prefixes[of 2 j # a] by auto
have phi-j:  $\varphi$  j = j # a ⊙ ↑∞
proof (rule prepend-eqI)
  show  $\bigwedge x. x < \text{length } (j \# a) \implies \varphi j x \downarrow = (j \# a) ! x$ 
    using assms(1,3) vs vs-def  $\langle 0 < z \rangle$ 
      length-prefix[of  $\psi j z - 1$ ]
      prefix-nth[of - -  $\psi j$ ]
      psi-at-0[of j]
      case-two-psi-longest-prefix[OF - z-def]
      longest-prefix[OF - z-def]
    by (metis One-nat-def Suc-pred option.collapse)
  show  $\bigwedge x. \varphi j (\text{length } (j \# a) + x) \uparrow$ 
    using assms(3) vs-def
    by (simp add: vs assms(1) case-two-psi-longest-prefix z-def)
qed
moreover have  $\varphi k \in V\text{-constotal-2}$ 
proof (intro V-constotal-2I[of - j a k])
  show  $\varphi k = j \# a @ [k] \odot 0^\infty$ 
    using phi-k .
  show 2 ≤ j
    using  $\langle 2 \leq j \rangle$  .
  show 2 ≤ k
    using  $\langle 2 \leq k \rangle$  .
  show  $\forall i < \text{length } a. a ! i \leq 1$ 
proof (rule allI, rule impI)
  fix i assume i: i < length a
  then have Suc i < z
    using z-def vs-def length-prefix  $\langle 0 < z \rangle$  vs
    by (metis One-nat-def Suc-mono Suc-pred length-Cons)
  have a ! i = vs ! (Suc i)
    using vs by simp
  also have ... = the ( $\psi j$  (Suc i))
    using vs-def vs i length-Cons length-prefix prefix-nth
    by (metis Suc-mono)
  finally show a ! i ≤ 1
    using case-two-psi-longest-prefix  $\langle \text{Suc } i < z \rangle$  z-def
    by (metis assms(1) less-or-eq-imp-le not-le-imp-less not-one-less-zero
      option.sel zero-less-Suc)
qed
qed (auto simp add: phi-j)
then have  $\varphi k \in V\text{-constotal}$ 
  using V-constotal-def by auto
moreover have  $\neg \text{learn-total } \varphi \{ \varphi k \} s$ 
proof -
  have  $\varphi k \in \mathcal{R}$ 
    by (simp add: phi-k almost0-in-R1)
  moreover have  $\bigwedge x. x < \text{longest-prefix } j \implies \varphi k x = \psi j x$ 
    using phi-k vs-def z-def length-prefix phi-j prepend-associative prepend-at-less

```

```

    by (metis One-nat-def Suc-pred ⟨0 < z⟩ ⟨vs = j # a⟩ append-Cons assms(3))
  ultimately show ?thesis
    using case-two-not-learn[OF assms(1)] by simp
qed
ultimately show ¬ learn-total  $\varphi$  V-constotal s
  using learn-total-closed-subseteq by auto
qed

```

The strategy S does not learn V in either case.

```

lemma not-learn-total-V: ¬ learn-total  $\varphi$  V-constotal s
proof -
  obtain j where j ≥ 2  $\varphi$  j =  $\psi$  j
  using kleene-fixed-point psi-in-P2 by auto
  then show ?thesis
    using case-one-not-learn-V learn-total-def case-two-not-learn-V
    by (cases case-two j) auto
qed
end

```

```

lemma V-not-in-TOTAL: V-constotal  $\notin$  TOTAL
proof (rule ccontr)
  assume ¬ V-constotal  $\notin$  TOTAL
  then have V-constotal  $\in$  TOTAL by simp
  then have V-constotal  $\in$  TOTAL-wrt  $\varphi$ 
    by (simp add: TOTAL-wrt-phi-eq-TOTAL)
  then obtain s where learn-total  $\varphi$  V-constotal s
    using TOTAL-wrt-def by auto
  then obtain s' where s': s'  $\in$   $\mathcal{R}$  learn-total  $\varphi$  V-constotal s'
    using lemma-R-for-TOTAL-simple by blast
  then interpret total-cons s'
    by (simp add: total-cons-def)
  have ¬ learn-total  $\varphi$  V-constotal s'
    by (simp add: not-learn-total-V)
  with s'(2) show False by simp
qed

```

```

lemma TOTAL-neq-CONS: TOTAL  $\neq$  CONS
  using V-not-in-TOTAL V-in-CONS CONS-def by auto

```

The main result of this section:

```

theorem TOTAL-subset-CONS: TOTAL  $\subset$  CONS
  using TOTAL-subseteq-CONS TOTAL-neq-CONS by simp

```

end

2.11 \mathcal{R} is not in BC

```

theory R1-BC
  imports Lemma-R
    CP-FIN-NUM
begin

```

We show that $U_0 \cup V_0$ is not in BC, which implies $\mathcal{R} \notin BC$.

The proof is by contradiction. Assume there is a strategy S learning $U_0 \cup V_0$ behaviorally correct in the limit with respect to our standard Gödel numbering φ . Thanks to Lemma R for BC we can assume S to be total. Then we construct a function in $U_0 \cup V_0$ for which S fails.

As usual, there is a computable process building prefixes of functions ψ_j . For every j it starts with the singleton prefix $b = [j]$ and computes the next prefix from a given prefix b as follows:

1. Simulate $\varphi_{S(b0^k)}(|b| + k)$ for increasing k for an increasing number of steps.
2. Once a k with $\varphi_{S(b0^k)}(|b| + k) = 0$ is found, extend the prefix by $0^k 1$.

There is always such a k because by assumption S learns $b0^\infty \in U_0$ and thus outputs a hypothesis for $b0^\infty$ on almost all of its prefixes. Therefore for almost all prefixes of the form $b0^k$, we have $\varphi_{S(b0^k)} = b0^\infty$ and hence $\varphi_{S(b0^k)}(|b| + k) = 0$. But Step 2 constructs ψ_j such that $\psi_j(|b| + k) = 1$. Therefore S does not hypothesize ψ_j on the prefix $b0^k$ of ψ_j . And since the process runs forever, S outputs infinitely many incorrect hypotheses for ψ_j and thus does not learn ψ_j .

Applying Kleene's fixed-point theorem to $\psi \in \mathcal{R}^2$ yields a j with $\varphi_j = \psi_j$ and thus $\psi_j \in V_0$. But S does not learn any ψ_j , contradicting our assumption.

The result $\mathcal{R} \notin BC$ can be obtained more directly by running the process with the empty prefix, thereby constructing only one function instead of a numbering. This function is in \mathcal{R} , and S fails to learn it by the same reasoning as above. The stronger statement about $U_0 \cup V_0$ will be exploited in Section 2.12.

In the following locale the assumption that S learns U_0 suffices for analyzing the process. However, in order to arrive at the desired contradiction this assumption is too weak because the functions built by the process are not in U_0 .

locale *r1-bc* =

fixes $s :: \text{partial1}$

assumes *s-in-R1*: $s \in \mathcal{R}$ **and** *s-learn-U0*: *learn-bc* φ U_0 s

begin

lemma *s-learn-prenum*: $\bigwedge b. \text{learn-bc } \varphi \{ \text{prenum } b \} s$

using *s-learn-U0* *U0-altdef* *learn-bc-closed-subseteq* **by** *blast*

A *ref* for the strategy:

definition *r-s* :: *ref* **where**

$r-s \equiv \text{SOME } rs. \text{refn } 1 \text{ } rs \wedge \text{total } rs \wedge s = (\lambda x. \text{eval } rs \ [x])$

lemma *r-s-refn* [*simp*]: *refn* 1 *r-s*

and *r-s-total*: $\bigwedge x. \text{eval } r-s \ [x] \downarrow$

and *eval-r-s*: $\bigwedge x. s \ x = \text{eval } r-s \ [x]$

using *r-s-def* *R1-SOME*[*OF* *s-in-R1*, *of* *r-s*] **by** *simp-all*

We begin with the function that finds the k from Step 1 of the construction of ψ .

definition *r-find-k* \equiv

let $k = \text{Cn } 2 \text{ } r\text{-pdec1} \ [\text{Id } 2 \ 0];$

$r = \text{Cn } 2 \text{ } r\text{-result1}$

$[\text{Cn } 2 \text{ } r\text{-pdec2} \ [\text{Id } 2 \ 0],$

$\text{Cn } 2 \text{ } r-s \ [\text{Cn } 2 \text{ } r\text{-append-zeros} \ [\text{Id } 2 \ 1, \ k]],$

$\text{Cn } 2 \text{ } r\text{-add} \ [\text{Cn } 2 \text{ } r\text{-length} \ [\text{Id } 2 \ 1], \ k]]$

in Cn 1 r-pdec1 [Mn 1 (Cn 2 r-eq [r, r-constn 1 1])]

lemma *r-find-k-recfn* [simp]: *recfn 1 r-find-k*
unfolding *r-find-k-def* **by** (*simp add: Let-def*)

There is always a suitable k , since the strategy learns $b0^\infty$ for all b .

lemma *learn-bc-prenum-eventually-zero*:

$\exists k. \varphi (\text{the } (s \text{ (e-append-zeros } b \ k))) \text{ (e-length } b + k) \downarrow = 0$

proof –

let $?f = \text{prenum } b$

have $\exists n \geq \text{e-length } b. \varphi (\text{the } (s \text{ (?f } \triangleright \ n))) = ?f$

using *learn-bcE s-learn-prenum* **by** (*meson le-cases singletonI*)

then obtain n **where** $n: n \geq \text{e-length } b \ \varphi (\text{the } (s \text{ (?f } \triangleright \ n))) = ?f$

by *auto*

define k **where** $k = \text{Suc } n - \text{e-length } b$

let $?e = \text{e-append-zeros } b \ k$

have $\text{len}: \text{e-length } ?e = \text{Suc } n$

using *k-def n e-append-zeros-length* **by** *simp*

have $?f \triangleright n = ?e$

proof –

have $\text{e-length } ?e > 0$

using *len n(1)* **by** *simp*

moreover have $?f \ x \downarrow = \text{e-nth } ?e \ x$ **for** x

proof (*cases* $x < \text{e-length } b$)

case *True*

then show *?thesis* **using** *e-nth-append-zeros* **by** *simp*

next

case *False*

then have $?f \ x \downarrow = 0$ **by** *simp*

moreover from *False* **have** $\text{e-nth } ?e \ x = 0$

using *e-nth-append-zeros-big* **by** *simp*

ultimately show *?thesis* **by** *simp*

qed

ultimately show *?thesis* **using** *initI[of ?e] len* **by** *simp*

qed

with *n(2)* **have** $\varphi (\text{the } (s \ ?e)) = ?f$ **by** *simp*

then have $\varphi (\text{the } (s \ ?e)) \text{ (e-length } ?e) \downarrow = 0$

using *len n(1)* **by** *auto*

then show *?thesis* **using** *e-append-zeros-length* **by** *auto*

qed

lemma *if-eq-eq*: (*if* $v = 1$ *then* $(0 :: \text{nat})$ *else* 1) = $0 \implies v = 1$

by *presburger*

lemma *r-find-k*:

shows *eval r-find-k [b]* \downarrow

and *let* $k = \text{the } (\text{eval } r\text{-find-k } [b])$

in $\varphi (\text{the } (s \text{ (e-append-zeros } b \ k))) \text{ (e-length } b + k) \downarrow = 0$

proof –

let $?k = \text{Cn } 2 \ r\text{-pdec1 } [Id \ 2 \ 0]$

let $?argt = \text{Cn } 2 \ r\text{-pdec2 } [Id \ 2 \ 0]$

let $?argi = \text{Cn } 2 \ r\text{-s } [\text{Cn } 2 \ r\text{-append-zeros } [Id \ 2 \ 1], ?k]$

let $?argx = \text{Cn } 2 \ r\text{-add } [\text{Cn } 2 \ r\text{-length } [Id \ 2 \ 1], ?k]$

let $?r = \text{Cn } 2 \ r\text{-result1 } [?argt, ?argi, ?argx]$

define f **where** $f \equiv$

let $k = \text{Cn } 2 \ r\text{-pdec1 } [Id \ 2 \ 0];$

```

    r = Cn 2 r-result1
      [Cn 2 r-pdec2 [Id 2 0],
       Cn 2 r-s [Cn 2 r-append-zeros [Id 2 1, k]],
       Cn 2 r-add [Cn 2 r-length [Id 2 1], k]]
    in Cn 2 r-eq [r, r-constn 1 1]
  then have recfn 2 f by (simp add: Let-def)
  have total r-s
    by (simp add: r-s-total totalI1)
  then have total f
    unfolding f-def using Cn-total Mn-free-imp-total by (simp add: Let-def)

  have eval ?argi [z, b] = s (e-append-zeros b (pdec1 z)) for z
    using r-append-zeros <recfn 2 f> eval-r-s by auto
  then have eval ?argi [z, b] ↓= the (s (e-append-zeros b (pdec1 z))) for z
    using eval-r-s r-s-total by simp
  moreover have recfn 2 ?r using <recfn 2 f> by auto
  ultimately have r: eval ?r [z, b] =
    eval r-result1 [pdec2 z, the (s (e-append-zeros b (pdec1 z))), e-length b + pdec1 z]
    for z
    by simp
  then have f: eval f [z, b] ↓= (if the (eval ?r [z, b]) = 1 then 0 else 1) for z
    using f-def <recfn 2 f> prim-recfn-total by (auto simp add: Let-def)

  have ∃ k. φ (the (s (e-append-zeros b k))) (e-length b + k) ↓= 0
    using s-learn-prenum learn-bc-prenum-eventually-zero by auto
  then obtain k where φ (the (s (e-append-zeros b k))) (e-length b + k) ↓= 0
    by auto
  then obtain t where eval r-result1 [t, the (s (e-append-zeros b k)), e-length b + k] ↓= Suc 0
    using r-result1-converg-phi(1) by blast
  then have t: eval r-result1 [t, the (s (e-append-zeros b k)), e-length b + k] ↓= Suc 0
    by simp

  let ?z = prod-encode (k, t)
  have eval ?r [?z, b] ↓= Suc 0
    using t r by (metis fst-conv prod-encode-inverse snd-conv)
  with f have fzb: eval f [?z, b] ↓= 0 by simp
  moreover have eval (Mn 1 f) [b] =
    (if (∃ z. eval f ([z, b]) ↓= 0)
     then Some (LEAST z. eval f [z, b] ↓= 0)
     else None)
    using eval-Mn-total[of 1 f [b]] <total f> <recfn 2 f> by simp
  ultimately have mn1f: eval (Mn 1 f) [b] ↓= (LEAST z. eval f [z, b] ↓= 0)
    by auto
  with fzb have eval f [the (eval (Mn 1 f) [b]), b] ↓= 0 (is eval f [?zz, b] ↓= 0)
    using <total f> <recfn 2 f> LeastI-ex[of %z. eval f [z, b] ↓= 0] by auto
  moreover have eval f [?zz, b] ↓= (if the (eval ?r [?zz, b]) = 1 then 0 else 1)
    using f by simp
  ultimately have (if the (eval ?r [?zz, b]) = 1 then (0 :: nat) else 1) = 0 by auto
  then have the (eval ?r [?zz, b]) = 1
    using if-eq-eq[of the (eval ?r [?zz, b])] by simp
  then have
    eval r-result1
      [pdec2 ?zz, the (s (e-append-zeros b (pdec1 ?zz))), e-length b + pdec1 ?zz] ↓=
      1
    using r r-result1-total r-result1-prim totalE
    by (metis length-Cons list.size(3) numeral-3-eq-3 option.collapse)

```

then have *: φ (the (s (e-append-zeros b (pdec1 ?zz)))) (e-length b + pdec1 ?zz) $\downarrow = 0$
by (simp add: r-result1-some-phi)

define Mn1f **where** Mn1f = Mn 1 f

then have eval Mn1f [b] $\downarrow = ?zz$

using mn1f **by** auto

moreover have recfn 1 (Cn 1 r-pdec1 [Mn1f])

using <recfn 2 f> Mn1f-def **by** simp

ultimately have eval (Cn 1 r-pdec1 [Mn1f]) [b] = eval r-pdec1 [the (eval (Mn1f) [b])]

by auto

then have eval (Cn 1 r-pdec1 [Mn1f]) [b] = eval r-pdec1 [?zz]

using Mn1f-def **by** blast

then have 1: eval (Cn 1 r-pdec1 [Mn1f]) [b] $\downarrow =$ pdec1 ?zz

by simp

moreover have recfn 1 (Cn 1 S [Cn 1 r-pdec1 [Mn1f]])

using <recfn 2 f> Mn1f-def **by** simp

ultimately have eval (Cn 1 S [Cn 1 r-pdec1 [Mn1f]]) [b] =

eval S [the (eval (Cn 1 r-pdec1 [Mn1f]) [b])]

by simp

then have eval (Cn 1 S [Cn 1 r-pdec1 [Mn1f]]) [b] = eval S [pdec1 ?zz]

using 1 **by** simp

then have eval (Cn 1 S [Cn 1 r-pdec1 [Mn1f]]) [b] $\downarrow =$ Suc (pdec1 ?zz)

by simp

moreover have eval r-find-k [b] = eval (Cn 1 r-pdec1 [Mn1f]) [b]

unfolding r-find-k-def Mn1f-def f-def **by** metis

ultimately have r-find-ksb: eval r-find-k [b] $\downarrow =$ pdec1 ?zz

using 1 **by** simp

then show eval r-find-k [b] \downarrow **by** simp-all

from r-find-ksb **have** the (eval r-find-k [b]) = pdec1 ?zz

by simp

moreover have φ (the (s (e-append-zeros b (pdec1 ?zz)))) (e-length b + pdec1 ?zz) $\downarrow = 0$

using * **by** simp

ultimately show let k = the (eval r-find-k [b])

in φ (the (s (e-append-zeros b k))) (e-length b + k) $\downarrow = 0$

by simp

qed

lemma r-find-k-total: total r-find-k

by (simp add: s-learn-prenum r-find-k(1) totalI1)

The following function represents one iteration of the process.

abbreviation r-next \equiv

Cn 3 r-snoc [Cn 3 r-append-zeros [Id 3 1, Cn 3 r-find-k [Id 3 1]], r-constn 2 1]

Using r-next we define the function r-prefixes that computes the prefix after every iteration of the process.

definition r-prefixes :: recf **where**

r-prefixes \equiv Pr 1 r-singleton-encode r-next

lemma r-prefixes-recfn: recfn 2 r-prefixes

unfolding r-prefixes-def **by** simp

lemma r-prefixes-total: total r-prefixes

proof –

```

have recfn 3 r-next by simp
then have total r-next
  using  $\langle \text{recfn } 3 \text{ r-next} \rangle$  r-find-k-total Cn-total Mn-free-imp-total by auto
then show ?thesis
  by (simp add: Mn-free-imp-total Pr-total r-prefixes-def)
qed

```

```

lemma r-prefixes-0: eval r-prefixes [0, j] ↓ = list-encode [j]
  unfolding r-prefixes-def by simp

```

```

lemma r-prefixes-Suc:
  eval r-prefixes [Suc n, j] ↓ =
    (let b = the (eval r-prefixes [n, j]))
    in e-snoc (e-append-zeros b (the (eval r-find-k [b]))) 1

```

proof –

```

have recfn 3 r-next by simp
then have total r-next
  using  $\langle \text{recfn } 3 \text{ r-next} \rangle$  r-find-k-total Cn-total Mn-free-imp-total by auto
have eval-next: eval r-next [t, v, j] ↓ =
  e-snoc (e-append-zeros v (the (eval r-find-k [v]))) 1
  for t v j
  using r-find-k-total  $\langle \text{recfn } 3 \text{ r-next} \rangle$  r-append-zeros by simp
then have eval r-prefixes [Suc n, j] = eval r-next [n, the (eval r-prefixes [n, j]), j]
  using r-prefixes-total by (simp add: r-prefixes-def)
then show eval r-prefixes [Suc n, j] ↓ =
  (let b = the (eval r-prefixes [n, j]))
  in e-snoc (e-append-zeros b (the (eval r-find-k [b]))) 1
  using eval-next by metis
qed

```

Since *r-prefixes* is total, we can get away with introducing a total function.

```

definition prefixes :: nat ⇒ nat ⇒ nat where
  prefixes j t ≡ the (eval r-prefixes [t, j])

```

```

lemma prefixes-Suc:
  prefixes j (Suc t) =
    e-snoc (e-append-zeros (prefixes j t) (the (eval r-find-k [prefixes j t]))) 1
  unfolding prefixes-def using r-prefixes-Suc by (simp-all add: Let-def)

```

```

lemma prefixes-Suc-length:
  e-length (prefixes j (Suc t)) =
    Suc (e-length (prefixes j t) + the (eval r-find-k [prefixes j t]))
  using e-append-zeros-length prefixes-Suc by simp

```

```

lemma prefixes-length-mono: e-length (prefixes j t) < e-length (prefixes j (Suc t))
  using prefixes-Suc-length by simp

```

```

lemma prefixes-length-mono': e-length (prefixes j t) ≤ e-length (prefixes j (t + d))

```

proof (*induction d*)

case *0*

then show *?case* **by** *simp*

next

case (*Suc d*)

then show *?case* **using** *prefixes-length-mono le-less-trans* **by** *fastforce*

qed

lemma *prefixes-length-lower-bound*: $e\text{-length}(\text{prefixes } j \ t) \geq \text{Suc } t$
proof (*induction t*)
 case 0
 then show ?*case* **by** (*simp add: prefixes-def r-prefixes-0*)
next
 case (*Suc t*)
 moreover have $\text{Suc } (e\text{-length}(\text{prefixes } j \ t)) \leq e\text{-length}(\text{prefixes } j \ (\text{Suc } t))$
 using *prefixes-length-mono* **by** (*simp add: Suc-leI*)
 ultimately show ?*case* **by** *simp*
qed

lemma *prefixes-Suc-nth*:
 assumes $x < e\text{-length}(\text{prefixes } j \ t)$
 shows $e\text{-nth}(\text{prefixes } j \ t) \ x = e\text{-nth}(\text{prefixes } j \ (\text{Suc } t)) \ x$
proof –
 define *k* **where** $k = \text{the } (e\text{-eval } r\text{-find-}k \ [\text{prefixes } j \ t])$
 let ?*u* = $e\text{-append-zeros}(\text{prefixes } j \ t) \ k$
 have $\text{prefixes } j \ (\text{Suc } t) =$
 $e\text{-snoc } (e\text{-append-zeros}(\text{prefixes } j \ t) \ (\text{the } (e\text{-eval } r\text{-find-}k \ [\text{prefixes } j \ t]))) \ 1$
 using *prefixes-Suc* **by** *simp*
 with *k-def* **have** $\text{prefixes } j \ (\text{Suc } t) = e\text{-snoc } ?u \ 1$
 by *simp*
 then have $e\text{-nth}(\text{prefixes } j \ (\text{Suc } t)) \ x = e\text{-nth}(e\text{-snoc } ?u \ 1) \ x$
 by *simp*
 moreover have $x < e\text{-length } ?u$
 using *assms e-append-zeros-length* **by** *auto*
 ultimately have $e\text{-nth}(\text{prefixes } j \ (\text{Suc } t)) \ x = e\text{-nth } ?u \ x$
 using *e-nth-snoc-small* **by** *simp*
 moreover have $e\text{-nth } ?u \ x = e\text{-nth}(\text{prefixes } j \ t) \ x$
 using *assms e-nth-append-zeros* **by** *simp*
 ultimately show $e\text{-nth}(\text{prefixes } j \ t) \ x = e\text{-nth}(\text{prefixes } j \ (\text{Suc } t)) \ x$
 by *simp*
qed

lemma *prefixes-Suc-last*: $e\text{-nth}(\text{prefixes } j \ (\text{Suc } t)) \ (e\text{-length}(\text{prefixes } j \ (\text{Suc } t)) - 1) = 1$
using *prefixes-Suc* **by** *simp*

lemma *prefixes-le-nth*:
 assumes $x < e\text{-length}(\text{prefixes } j \ t)$
 shows $e\text{-nth}(\text{prefixes } j \ t) \ x = e\text{-nth}(\text{prefixes } j \ (t + d)) \ x$
proof (*induction d*)
 case 0
 then show ?*case* **by** *simp*
next
 case (*Suc d*)
 have $x < e\text{-length}(\text{prefixes } j \ (t + d))$
 using *s-learn-prenum assms prefixes-length-mono'*
 by (*simp add: less-eq-Suc-le order-trans-rules(23)*)
 then have $e\text{-nth}(\text{prefixes } j \ (t + d)) \ x = e\text{-nth}(\text{prefixes } j \ (t + \text{Suc } d)) \ x$
 using *prefixes-Suc-nth* **by** *simp*
 with *Suc* **show** ?*case* **by** *simp*
qed

The numbering ψ is defined via *prefixes*.

definition *psi* :: *partial2* ($\langle \psi \rangle$) **where**
 $\psi \ j \ x \equiv \text{Some } (e\text{-nth}(\text{prefixes } j \ (\text{Suc } x)) \ x)$

lemma *psi-in-R2*: $\psi \in \mathcal{R}^2$

proof

define r **where** $r \equiv Cn\ 2\ r\text{-nth}\ [Cn\ 2\ r\text{-prefixes}\ [Cn\ 2\ S\ [Id\ 2\ 1],\ Id\ 2\ 0],\ Id\ 2\ 1]$
then have $recfn\ 2\ r$
using $r\text{-prefixes-recfn}$ **by** *simp*
then have $eval\ r\ [j,\ x] \downarrow = e\text{-nth}\ (prefixes\ j\ (Suc\ x))\ x$ **for** $j\ x$
unfolding $r\text{-def}\ prefixes\text{-def}$ **using** $r\text{-prefixes-total}\ r\text{-prefixes-recfn}\ e\text{-nth}$ **by** *simp*
then have $eval\ r\ [j,\ x] = \psi\ j\ x$ **for** $j\ x$
unfolding $psi\text{-def}$ **by** *simp*
then show $\psi \in \mathcal{P}^2$
using $\langle recfn\ 2\ r \rangle$ **by** *auto*
show $total2\ \psi$
unfolding $psi\text{-def}$ **by** *auto*

qed

lemma *psi-eq-nth-prefixes*:

assumes $x < e\text{-length}\ (prefixes\ j\ t)$
shows $\psi\ j\ x \downarrow = e\text{-nth}\ (prefixes\ j\ t)\ x$

proof (*cases* $Suc\ x < t$)

case *True*

have $x \leq e\text{-length}\ (prefixes\ j\ x)$
using $prefixes\text{-length-lower-bound}$ **by** (*simp add: Suc-leD*)
also have $\dots < e\text{-length}\ (prefixes\ j\ (Suc\ x))$
using $prefixes\text{-length-mono}\ s\text{-learn-prenum}$ **by** *simp*
finally have $x < e\text{-length}\ (prefixes\ j\ (Suc\ x))$.
with *True* **have** $e\text{-nth}\ (prefixes\ j\ (Suc\ x))\ x = e\text{-nth}\ (prefixes\ j\ t)\ x$
using $prefixes\text{-le-nth}[of\ x\ j\ Suc\ x\ t - Suc\ x]$ **by** *simp*
then show *?thesis* **using** $psi\text{-def}$ **by** *simp*

next

case *False*

then have $e\text{-nth}\ (prefixes\ j\ (Suc\ x))\ x = e\text{-nth}\ (prefixes\ j\ t)\ x$
using $prefixes\text{-le-nth}[of\ x\ j\ t\ Suc\ x - t]$ *assms* **by** *simp*
then show *?thesis* **using** $psi\text{-def}$ **by** *simp*

qed

lemma *psi-at-0*: $\psi\ j\ 0 \downarrow = j$

using $psi\text{-eq-nth-prefixes}[of\ 0\ j\ 0]$ $prefixes\text{-length-lower-bound}[of\ 0\ j]$
by (*simp add: prefixes-def r-prefixes-0*)

The prefixes output by the process $prefixes\ j$ are indeed prefixes of ψ_j .

lemma *prefixes-init-psi*: $\psi\ j \triangleright (e\text{-length}\ (prefixes\ j\ (Suc\ t)) - 1) = prefixes\ j\ (Suc\ t)$

proof (*rule* $initI[of\ prefixes\ j\ (Suc\ t)]$)

let $?e = prefixes\ j\ (Suc\ t)$

show $e\text{-length}\ ?e > 0$

using $prefixes\text{-length-lower-bound}[of\ Suc\ t\ j]$ **by** *auto*

show $\bigwedge x. x < e\text{-length}\ ?e \implies \psi\ j\ x \downarrow = e\text{-nth}\ ?e\ x$

using $prefixes\text{-Suc-nth}\ psi\text{-eq-nth-prefixes}$ **by** *simp*

qed

Every prefix of ψ_j generated by the process $prefixes\ j$ (except for the initial one) is of the form $b0^k1$. But k is chosen such that $\varphi_{S(b0^k)}(|b| + k) = 0 \neq 1 = b0^k1_{|b|+k}$. Therefore the hypothesis $S(b0^k)$ is incorrect for ψ_j .

lemma *hyp-wrong-at-last*:

$\varphi\ (the\ (s\ (e\text{-butlast}\ (prefixes\ j\ (Suc\ t))))\ (e\text{-length}\ (prefixes\ j\ (Suc\ t)) - 1) \neq$

ψj ($e\text{-length}$ ($\text{prefixes } j$ ($\text{Suc } t$)) - 1)
 (is $?lhs \neq ?rhs$)
proof -
 let $?b = \text{prefixes } j t$
 let $?k = \text{the } (eval\ r\text{-find-}k\ [?b])$
 let $?x = e\text{-length } (\text{prefixes } j\ (\text{Suc } t)) - 1$
 have $e\text{-butlast } (\text{prefixes } j\ (\text{Suc } t)) = e\text{-append-zeros } ?b\ ?k$
 using $s\text{-learn-prenum } \text{prefixes-Suc}$ **by** simp
 then have $?lhs = \varphi$ ($\text{the } (s\ (e\text{-append-zeros } ?b\ ?k))$) $?x$
 by simp
 moreover have $?x = e\text{-length } ?b + ?k$
 using $\text{prefixes-Suc-length}$ **by** simp
 ultimately have $?lhs = \varphi$ ($\text{the } (s\ (e\text{-append-zeros } ?b\ ?k))$) ($e\text{-length } ?b + ?k$)
 by simp
 then have $?lhs \downarrow = 0$
 using $r\text{-find-}k(2)\ r\text{-s-total } s\text{-learn-prenum}$ **by** metis
 moreover have $?x < e\text{-length } (\text{prefixes } j\ (\text{Suc } t))$
 using $\text{prefixes-length-lower-bound } le\text{-less-trans } \text{linorder-not-le } s\text{-learn-prenum}$
 by fastforce
 ultimately have $?rhs \downarrow = e\text{-nth } (\text{prefixes } j\ (\text{Suc } t))\ ?x$
 using $\text{psi-eg-nth-prefixes}$ [of $?x\ j\ \text{Suc } t$] **by** simp
 moreover have $e\text{-nth } (\text{prefixes } j\ (\text{Suc } t))\ ?x = 1$
 using $\text{prefixes-Suc } \text{prefixes-Suc-last}$ **by** simp
 ultimately have $?rhs \downarrow = 1$ **by** simp
 with $\langle ?lhs \downarrow = 0 \rangle$ **show** $?thesis$ **by** simp
qed

corollary hyp-wrong : φ ($\text{the } (s\ (e\text{-butlast } (\text{prefixes } j\ (\text{Suc } t))))$) $\neq \psi j$
 using hyp-wrong-at-last [of $j\ t$] **by** auto

For all j , the strategy S outputs infinitely many wrong hypotheses for ψ_j

lemma $\text{infinite-hyp-wrong}$: $\exists m > n. \varphi$ ($\text{the } (s\ (\psi\ j\ \triangleright\ m))$) $\neq \psi j$

proof -
 let $?b = \text{prefixes } j\ (\text{Suc } (\text{Suc } n))$
 let $?bb = e\text{-butlast } ?b$
 have $\text{len-b: } e\text{-length } ?b > \text{Suc } (\text{Suc } n)$
 using $\text{prefixes-length-lower-bound}$ **by** ($\text{simp add: } \text{Suc-le-lessD}$)
 then have $\text{len-bb: } e\text{-length } ?bb > \text{Suc } n$ **by** simp
 define m where $m = e\text{-length } ?bb - 1$
 with len-bb have $m > n$ **by** simp
 have $\psi j \triangleright m = ?bb$
proof -
 have $\psi j \triangleright (e\text{-length } ?b - 1) = ?b$
 using prefixes-init-psi **by** simp
 then have $\psi j \triangleright (e\text{-length } ?b - 2) = ?bb$
 using $\text{init-butlast-init } \text{psi-in-R2 } \text{R2-proj-R1 } \text{R1-imp-total1 } \text{len-bb } \text{length-init}$
 by ($\text{metis } \text{Suc-1 } \text{diff-diff-left } \text{length-butlast } \text{length-greater-0-conv}$
 $\text{list.size}(3)\ \text{list-decode-encode } \text{not-less0 } \text{plus-1-eq-Suc}$)
 then **show** $?thesis$ **by** ($\text{metis } \text{diff-Suc-1 } \text{length-init } m\text{-def}$)
qed
 moreover have φ ($\text{the } (s\ ?bb)$) $\neq \psi j$
 using hyp-wrong **by** simp
 ultimately have φ ($\text{the } (s\ (\psi\ j\ \triangleright\ m))$) $\neq \psi j$
 by simp
 with $\langle m > n \rangle$ **show** $?thesis$ **by** auto
qed

lemma *U0-V0-not-learn-bc*: $\neg \text{learn-bc } \varphi (U_0 \cup V_0) s$
proof –
 obtain *j* where *j*: $\varphi j = \psi j$
 using *R2-imp-P2 kleene-fixed-point psi-in-R2* **by** *blast*
 moreover **have** $\exists m > n. \varphi (\text{the } (s ((\psi j) \triangleright m))) \neq \psi j$ **for** *n*
 using *infinite-hyp-wrong[of - j]* **by** *simp*
 ultimately **have** $\neg \text{learn-bc } \varphi \{\psi j\} s$
 using *infinite-hyp-wrong-not-BC* **by** *simp*
 moreover **have** $\psi j \in V_0$
proof –
 have $\psi j \in \mathcal{R}$ (**is** $?f \in \mathcal{R}$)
 using *psi-in-R2* **by** *simp*
 moreover **have** $\varphi (\text{the } (?f 0)) = ?f$
 using *j psi-at-0[of j]* **by** *simp*
 ultimately **show** *?thesis* **by** (*simp add: V0-def*)
qed
 ultimately **show** $\neg \text{learn-bc } \varphi (U_0 \cup V_0) s$
 using *learn-bc-closed-subseteq* **by** *auto*
qed
end

lemma *U0-V0-not-in-BC*: $U_0 \cup V_0 \notin BC$
proof
 assume *in-BC*: $U_0 \cup V_0 \in BC$
 then **have** $U_0 \cup V_0 \in BC\text{-wrt } \varphi$
 using *BC-wrt-phi-eq-BC* **by** *simp*
 then **obtain** *s* where $\text{learn-bc } \varphi (U_0 \cup V_0) s$
 using *BC-wrt-def* **by** *auto*
 then **obtain** *s'* where *s'*: $s' \in \mathcal{R} \text{ learn-bc } \varphi (U_0 \cup V_0) s'$
 using *lemma-R-for-BC-simple* **by** *blast*
 then **have** *learn-U0*: $\text{learn-bc } \varphi U_0 s'$
 using *learn-bc-closed-subseteq[of \varphi U_0 \cup V_0 s']* **by** *simp*
 then **interpret** *r1-bc s'*
by (*simp add: r1-bc-def s'(1)*)
have $\neg \text{learn-bc } \varphi (U_0 \cup V_0) s'$
 using *learn-bc-closed-subseteq U0-V0-not-learn-bc* **by** *simp*
 with *s'(2)* **show** *False* **by** *simp*
qed

theorem *R1-not-in-BC*: $\mathcal{R} \notin BC$
proof –
have $U_0 \cup V_0 \subseteq \mathcal{R}$
 using *V0-def U0-in-NUM* **by** *auto*
 then **show** *?thesis*
 using *U0-V0-not-in-BC BC-closed-subseteq* **by** *auto*
qed

end

2.12 The union of classes

theory *Union*
imports *R1-BC TOTAL-CONS*

begin

None of the inference types introduced in this chapter are closed under union of classes. For all inference types except FIN this follows from *U0-V0-not-in-BC*.

lemma *not-closed-under-union*:

$\forall \mathcal{I} \in \{CP, TOTAL, CONS, LIM, BC\}. U_0 \in \mathcal{I} \wedge V_0 \in \mathcal{I} \wedge U_0 \cup V_0 \notin \mathcal{I}$

using *U0-in-CP U0-in-NUM V0-in-FIN*

FIN-subseteq-CP

NUM-subseteq-TOTAL

CP-subseteq-TOTAL

TOTAL-subseteq-CONS

CONS-subseteq-Lim

Lim-subseteq-BC

U0-V0-not-in-BC

by *blast*

In order to show the analogous result for FIN consider the classes $\{0^\infty\}$ and $\{0^n 10^\infty \mid n \in \mathbb{N}\}$. The former can be learned finitely by a strategy that hypothesizes 0^∞ for every input. The latter can be learned finitely by a strategy that waits for the 1 and hypothesizes the only function in the class with a 1 at that position. However, the union of both classes is not in FIN. This is because any FIN strategy has to hypothesize 0^∞ on some prefix of the form 0^n . But the strategy then fails for the function $0^n 10^\infty$.

lemma *singleton-in-FIN*: $f \in \mathcal{R} \implies \{f\} \in FIN$

proof –

assume $f \in \mathcal{R}$

then obtain i **where** $i: \varphi \ i = f$

using *phi-universal* **by** *blast*

define $s :: \text{partial1}$ **where** $s = (\lambda-. \text{Some} (\text{Suc } i))$

then have $s \in \mathcal{R}$

using *const-in-Prim1*[of *Suc i*] **by** *simp*

have *learn-fin* $\varphi \ \{f\} \ s$

proof (*intro learn-finI*)

show *environment* $\varphi \ \{f\} \ s$

using $\langle s \in \mathcal{R} \rangle \ \langle f \in \mathcal{R} \rangle$ **by** (*simp add: phi-in-P2*)

show $\exists i \ n_0. \varphi \ i = g \wedge (\forall n < n_0. s \ (g \triangleright n) \downarrow = 0) \wedge (\forall n \geq n_0. s \ (g \triangleright n) \downarrow = \text{Suc } i)$

if $g \in \{f\}$ **for** g

proof –

from that have $g = f$ **by** *simp*

then have $\varphi \ i = g$

using i **by** *simp*

moreover have $\forall n < 0. s \ (g \triangleright n) \downarrow = 0$ **by** *simp*

moreover have $\forall n \geq 0. s \ (g \triangleright n) \downarrow = \text{Suc } i$

using $s\text{-def}$ **by** *simp*

ultimately show *?thesis* **by** *auto*

qed

qed

then show $\{f\} \in FIN$ **using** *FIN-def* **by** *auto*

qed

definition *U-single* $:: \text{partial1 set where}$

$U\text{-single} \equiv \{(\lambda x. \text{if } x = n \text{ then Some } 1 \text{ else Some } 0) \mid n. n \in UNIV\}$

lemma *U-single-in-FIN*: $U\text{-single} \in FIN$

proof –

define $\psi :: \text{partial2}$ **where** $\psi \equiv \lambda n \ x. \text{if } x = n \text{ then Some } 1 \text{ else Some } 0$

```

have psi ∈ ℝ2
  using psi-def by (intro R2I[of Cn 2 r-not [r-eq]]) auto
define s :: partial1 where
  s ≡ λb. if findr b ↓= e-length b then Some 0 else Some (Suc (the (findr b)))
have s ∈ ℝ
proof (rule R1I)
  let ?r = Cn 1 r-ifeq [r-findr, r-length, Z, Cn 1 S [r-findr]]
  show recfn 1 ?r by simp
  show total ?r by auto
  show eval ?r [b] = s b for b
  proof -
    let ?b = the (findr b)
    have eval ?r [b] = (if ?b = e-length b then Some 0 else Some (Suc (?b)))
      using findr-total by simp
    then show eval ?r [b] = s b
      by (metis findr-total option.collapse option.inject s-def)
  qed
qed
have U-single ⊆ ℝ
proof
  fix f
  assume f ∈ U-single
  then obtain n where f = (λx. if x = n then Some 1 else Some 0)
    using U-single-def by auto
  then have f = psi n
    using psi-def by simp
  then show f ∈ ℝ
    using ⟨psi ∈ ℝ2⟩ by simp
qed
have learn-fin psi U-single s
proof (rule learn-finI)
  show environment psi U-single s
    using ⟨psi ∈ ℝ2⟩ ⟨s ∈ ℝ⟩ ⟨U-single ⊆ ℝ⟩ by simp
  show ∃ i n0. psi i = f ∧ (∀ n < n0. s (f ▷ n) ↓= 0) ∧ (∀ n ≥ n0. s (f ▷ n) ↓= Suc i)
    if f ∈ U-single for f
  proof -
    from that obtain i where i: f = (λx. if x = i then Some 1 else Some 0)
      using U-single-def by auto
    then have psi i = f
      using psi-def by simp
    moreover have ∀ n < i. s (f ▷ n) ↓= 0
      using i s-def findr-def by simp
    moreover have ∀ n ≥ i. s (f ▷ n) ↓= Suc i
  proof (rule allI, rule impI)
    fix n
    assume n ≥ i
    let ?e = init f n
    have ∃ i < e-length ?e. e-nth ?e i ≠ 0
      using ⟨n ≥ i⟩ i by simp
    then have less: the (findr ?e) < e-length ?e
      and nth-e: e-nth ?e (the (findr ?e)) ≠ 0
      using findr-ex by blast+
    then have s ?e ↓= Suc (the (findr ?e))
      using s-def by auto
    moreover have the (findr ?e) = i
      using nth-e less i by (metis length-init nth-init option.sel)
  qed
  qed

```

ultimately show $s \text{ ?}e \Downarrow = \text{Suc } i$ by *simp*
 qed
 ultimately show ?thesis by *auto*
 qed
 qed
 then show $U\text{-single} \in \text{FIN}$ using *FIN-def* by *blast*
 qed

lemma *zero-U-single-not-in-FIN*: $\{0^\infty\} \cup U\text{-single} \notin \text{FIN}$

proof

assume $\{0^\infty\} \cup U\text{-single} \in \text{FIN}$
 then obtain $\text{psi } s$ where *learn*: *learn-fin psi* $(\{0^\infty\} \cup U\text{-single}) s$
 using *FIN-def* by *blast*

then have *learn-fin psi* $\{0^\infty\} s$
 using *learn-fin-closed-subseteq* by *auto*

then obtain $i n_0$ where *i*:

$\text{psi } i = 0^\infty$
 $\forall n < n_0. s (0^\infty \triangleright n) \Downarrow = 0$
 $\forall n \geq n_0. s (0^\infty \triangleright n) \Downarrow = \text{Suc } i$
 using *learn-finE(2)* by *blast*

let $\text{?}f = \lambda x. \text{if } x = \text{Suc } n_0 \text{ then Some } 1 \text{ else Some } 0$

have $\text{?}f \neq 0^\infty$ by (*metis option.inject zero-neq-one*)

have $\text{?}f \in U\text{-single}$

using *U-single-def* by *auto*

then have *learn-fin psi* $\{\text{?}f\} s$

using *learn learn-fin-closed-subseteq* by *simp*

then obtain $j m_0$ where *j*:

$\text{psi } j = \text{?}f$
 $\forall n < m_0. s (\text{?}f \triangleright n) \Downarrow = 0$
 $\forall n \geq m_0. s (\text{?}f \triangleright n) \Downarrow = \text{Suc } j$
 using *learn-finE(2)* by *blast*

consider

$(\text{less}) m_0 < n_0 \mid (\text{eq}) m_0 = n_0 \mid (\text{gr}) m_0 > n_0$
 by *linarith*

then show *False*

proof (*cases*)

case *less*

then have $s (0^\infty \triangleright m_0) \Downarrow = 0$

using *i* by *simp*

moreover have $0^\infty \triangleright m_0 = \text{?}f \triangleright m_0$

using *less init-eqI[of m_0 ?f 0^\infty]* by *simp*

ultimately have $s (\text{?}f \triangleright m_0) \Downarrow = 0$ by *simp*

then show *False* using *j* by *simp*

next

case *eq*

then have $0^\infty \triangleright m_0 = \text{?}f \triangleright m_0$

using *init-eqI[of m_0 ?f 0^\infty]* by *simp*

then have $s (0^\infty \triangleright m_0) = s (\text{?}f \triangleright m_0)$ by *simp*

then have $i = j$

using *i j eq* by *simp*

then have $\text{psi } i = \text{psi } j$ by *simp*

then show *False* using $\langle \text{?}f \neq 0^\infty \rangle i j$ by *simp*

next

case *gr*

have $0^\infty \triangleright n_0 = \text{?}f \triangleright n_0$

using *init-eqI[of n_0 ?f 0^\infty]* by *simp*

moreover have $s (0^\infty \triangleright n_0) \Downarrow = \text{Suc } i$
using i **by** *simp*
moreover have $s (?f \triangleright n_0) \Downarrow = 0$
using $j \text{ gr}$ **by** *simp*
ultimately show *False* **by** *simp*
qed
qed

lemma *FIN-not-closed-under-union*: $\exists U V. U \in \text{FIN} \wedge V \in \text{FIN} \wedge U \cup V \notin \text{FIN}$

proof –

have $\{0^\infty\} \in \text{FIN}$
using *singleton-in-FIN const-in-Prim1* **by** *simp*
moreover have $U\text{-single} \in \text{FIN}$
using *U-single-in-FIN* **by** *simp*
ultimately show *?thesis*
using *zero-U-single-not-in-FIN* **by** *blast*
qed

In contrast to the inference types, NUM is closed under the union of classes. The total numberings that exist for each NUM class can be interleaved to produce a total numbering encompassing the union of the classes. To define the interleaving, modulo and division by two will be helpful.

definition *r-div2* \equiv

r-shrink
 $(Pr\ 1\ Z$
 $(Cn\ 3\ r\text{-ifle}$
 $[Cn\ 3\ r\text{-mul } [r\text{-constn } 2\ 2, Cn\ 3\ S\ [Id\ 3\ 0]], Id\ 3\ 2, Cn\ 3\ S\ [Id\ 3\ 1], Id\ 3\ 1]))$

lemma *r-div2-prim* [*simp*]: *prim-recfn 1 r-div2*

unfolding *r-div2-def* **by** *simp*

lemma *r-div2* [*simp*]: *eval r-div2 [n] $\Downarrow = n \text{ div } 2$*

proof –

let $?p = Pr\ 1\ Z$
 $(Cn\ 3\ r\text{-ifle}$
 $[Cn\ 3\ r\text{-mul } [r\text{-constn } 2\ 2, Cn\ 3\ S\ [Id\ 3\ 0]], Id\ 3\ 2, Cn\ 3\ S\ [Id\ 3\ 1], Id\ 3\ 1])$
have *eval ?p [i, n] $\Downarrow = \min (n \text{ div } 2) i$* **for** i
by (*induction i*) *auto*
then have *eval ?p [n, n] $\Downarrow = n \text{ div } 2$* **by** *simp*
then show *?thesis* **unfolding** *r-div2-def* **by** *simp*
qed

definition *r-mod2* $\equiv Cn\ 1\ r\text{-sub } [Id\ 1\ 0, Cn\ 1\ r\text{-mul } [r\text{-const } 2, r\text{-div2}]]$

lemma *r-mod2-prim* [*simp*]: *prim-recfn 1 r-mod2*

unfolding *r-mod2-def* **by** *simp*

lemma *r-mod2* [*simp*]: *eval r-mod2 [n] $\Downarrow = n \text{ mod } 2$*

unfolding *r-mod2-def* **using** *Rings.semiring-modulo-class.minus-mult-div-eq-mod*
by *auto*

lemma *NUM-closed-under-union*:

assumes $U \in \text{NUM}$ **and** $V \in \text{NUM}$

shows $U \cup V \in \text{NUM}$

proof –

```

from assms obtain psi-u psi-v where
  psi-u: psi-u  $\in \mathcal{R}^2 \wedge f. f \in U \implies \exists i. \text{psi-u } i = f$  and
  psi-v: psi-v  $\in \mathcal{R}^2 \wedge f. f \in V \implies \exists i. \text{psi-v } i = f$ 
  by fastforce
define psi where psi  $\equiv \lambda i. \text{if } i \text{ mod } 2 = 0 \text{ then } \text{psi-u } (i \text{ div } 2) \text{ else } \text{psi-v } (i \text{ div } 2)$ 
from psi-u(1) obtain u where u: recfn 2 u total u  $\wedge x y. \text{eval } u [x, y] = \text{psi-u } x y$ 
  by auto
from psi-v(1) obtain v where v: recfn 2 v total v  $\wedge x y. \text{eval } v [x, y] = \text{psi-v } x y$ 
  by auto
let ?r-psi = Cn 2 r-ifz
  [Cn 2 r-mod2 [Id 2 0],
   Cn 2 u [Cn 2 r-div2 [Id 2 0], Id 2 1],
   Cn 2 v [Cn 2 r-div2 [Id 2 0], Id 2 1]]
show ?thesis
proof (rule NUM-I[of psi])
  show psi  $\in \mathcal{R}^2$ 
  proof (rule R2I)
    show recfn 2 ?r-psi
      using u(1) v(1) by simp
    show eval ?r-psi  $[x, y] = \text{psi } x y$  for x y
      using u v psi-def prim-recfn-total R2-imp-total[OF psi-u(1)]
        R2-imp-total[OF psi-v(1)]
      by simp
    moreover have psi  $x y \downarrow$  for x y
      using psi-def psi-u(1) psi-v(1) by simp
    ultimately show total ?r-psi
      using  $\langle \text{recfn } 2 \text{ ?r-psi} \rangle \text{ totalI2}$  by simp
  qed
show  $\exists i. \text{psi } i = f$  if  $f \in U \cup V$  for f
proof (cases f  $\in U$ )
  case True
    then obtain j where psi-u j = f
      using psi-u(2) by auto
    then have psi (2 * j) = f
      using psi-def by simp
    then show ?thesis by auto
  next
  case False
    then have  $f \in V$ 
      using that by simp
    then obtain j where psi-v j = f
      using psi-v(2) by auto
    then have psi (Suc (2 * j)) = f
      using psi-def by simp
    then show ?thesis by auto
  qed
qed
qed
end

```

Bibliography

- [1] D. Angluin and C. H. Smith. Inductive inference. In *Encyclopedia of Artificial Intelligence*, pages 409–418. J. Wiley and Sons, New York, 1987.
- [2] J. M. Barzdin. Two theorems on the limiting synthesis of functions. In *Theory of Algorithms and Programs*, volume 1, pages 82–88. Latvian State University, Riga, 1974. In Russian.
- [3] J. M. Barzdin. Inductive inference of automata, functions and programs. In *Amer. Math. Soc. Transl.*, pages 107–122, 1977.
- [4] Y. M. Barzdin. Inductive inference of automata, functions and programs. In *Proceedings International Congress of Mathematics*, pages 455–460, 1974.
- [5] L. Blum and M. Blum. Toward a mathematical theory of inductive inference. *Inform. Control*, 28(2):125–155, June 1975.
- [6] J. Case and C. H. Smith. Comparison of identification criteria for machine inductive inference. *Theoret. Comput. Sci.*, 25:193–220, 1983.
- [7] R. Freivalds, E. B. Kinber, and R. Wiehagen. How inductive inference strategies discover their errors. *Inform. Comput.*, 118(2):208–226, 1995.
- [8] E. M. Gold. Limiting recursion. *J. Symbolic Logic*, 30:28–48, 1965.
- [9] E. M. Gold. Language identification in the limit. *Inform. Control*, 10(5):447–474, 1967.
- [10] K. P. Jantke and H.-R. Beick. Combining postulates of naturalness in inductive inference. *Elektronische Informationsverarbeitung und Kybernetik*, 17(8/9):465–484, 1981.
- [11] S. C. Kleene. Recursive predicates and quantifiers. *Trans. Amer. Math. Soc.*, 53(1):41–73, 1943.
- [12] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. The MIT Press, 2nd edition, 1987.
- [13] R. J. Solomonoff. A formal theory of inductive inference: Part 1. *Inform. Control*, 7:1–22, 1964.
- [14] R. J. Solomonoff. A formal theory of inductive inference: Part 2. *Inform. Control*, 7:224–254, 1964.
- [15] R. Wiehagen. Limes-Erkennung rekursiver Funktionen durch spezielle Strategien. *Journal of Information Processing and Cybernetics (EIK)*, 12:93–99, 1976.

- [16] R. Wiehagen and T. Zeugmann. Ignoring data may be the only way to learn efficiently. *J. of Experimental and Theoret. Artif. Intell.*, 6(1):131–144, 1994.
- [17] Wikipedia contributors. Kleene’s recursion theorem — Wikipedia, the free encyclopedia, 2020. [Online; accessed 28-March-2020].
- [18] J. Xu, X. Zhang, C. Urban, and S. J. C. Joosten. Universal turing machine. *Archive of Formal Proofs*, Feb. 2019. http://isa-afp.org/entries/Universal_Turing_Machine.html, Formal proof development.