Verifying a Decision Procedure for Pattern Completeness*

René Thiemann

University of Innsbruck, Austria

Akihisa Yamada

National Institute of Advanced Industrial Science and Technology, Japan

June 3, 2024

Abstract

Pattern completeness is the property that the left-hand sides of a functional program or term rewrite system cover all cases w.r.t. pattern matching. We verify a recent (abstract) decision procedure for pattern completeness that covers the general case, i.e., in particular without the usual restriction of left-linearity. In two refinement steps, we further develop an executable version of that abstract algorithm. On our example suite, this verified implementation is faster than other implementations that are based on alternative (unverified) approaches, including the complement algorithm, tree automata encodings, and even the pattern completeness check of the GHC Haskell compiler.

Contents

*This research was supported by the Austrian Science Fund (FWF) project I 5943.

1 Introduction

This AFP entry includes the formalization of a decision procedure [4] for pattern completeness. It also contains the setup for running the experiments of that paper, i.e., it contains

- a generator for example term rewrite systems and Haskell programs of varying size,
- a connection to an implementation of the complement algorithm [2] within the ground confluence prover AGCP [1], and
- a tree automata encoder of pattern completeness that is linked with the tree automata library FORT-h [3].

Note that some further glue code is required to run the experiments, which is not included in this submission. Here, we just include the glue code that was defined within Isabelle theories.

2 Pattern Completeness

Pattern-completeness is the question whether in a given program all terms of the form f(c1,..,cn) are matched by some lhs of the program, where here each ci is a constructor ground term and f is a defined symbol. This will be represented as a pattern problem of the shape (f(x1,...xn), lhs1, ..., lhsn) where the xi will represent arbitrary constructor terms.

3 A Set-Based Inference System to Decide Pattern Completeness

This theory contains an algorithm to decide whether pattern problems are complete. It represents the inference rules of the paper on the set-based level.

On this level we prove partial correctness and preservation of well-formed inputs, but not termination.

theory *Pattern-Completeness-Set*
  imports
    First-Order-Terms.Term-More
    Sorted-Terms.Sorted-Contexts
begin

3.1 Definition of Algorithm – Inference Rules

We first consider matching problems which are sets of term pairs. Note that in the term pairs the type of variables differ: Each left term has natural numbers (with sorts) as variables, so that it is easy to generate new variables, whereas each right term has arbitrary variables of type $'v$ without any further information. Then pattern problems are sets of matching problems, and we also have sets of pattern problems.

The suffix *-set* is used to indicate that here these problems are modeled via sets.

type-synonym $('f,'v,'s)$*match-problem-set* $=$ $(('f,nat \times 's)term \times ('f,'v)term)$ *set*

type-synonym $('f,'v,'s)$*pat-problem-set* $=$ $('f,'v,'s)$*match-problem-set set*
type-synonym $('f,'v,'s)$*pats-problem-set* $=$ $('f,'v,'s)$*pat-problem-set set*

abbreviation (*input*) *bottom* :: $('f,'v,'s)$*pats-problem-set* **where** *bottom* $\equiv \{\{\}\}$

definition *subst-left* :: $('f,nat \times 's)subst \Rightarrow (('f,nat \times 's)term \times ('f,'v)term) \Rightarrow (('f,nat \times 's)term \times ('f,'v)term)$ **where**
  subst-left $\tau = (\lambda(t,r).\ (t \cdot \tau,\ r))$

A function to compute for a variable x all substitution that instantiate x

by $c(x_n, ..., x_{n+a})$ where c is an constructor of arity a and n is a parameter that determines from where to start the numbering of variables.

definition τc :: *nat* ⇒ *nat* × *'s* ⇒ *'f* × *'s list* ⇒ (*'f*,*nat* × *'s*)*subst* **where**
τc *n x* = (λ(*f*,*ss*). *subst x* (*Fun f* (*map Var* (*zip* [*n* ..< *n* + *length ss*] *ss*))))

Compute the list of conflicting variables (Some list), or detect a clash (None)

fun *conflicts* :: (*'f*,*'v*)*term* ⇒ (*'f*,*'v*)*term* ⇒ *'v list option* **where**
conflicts (*Var x*) (*Var y*) = (*if x* = *y then Some* [] *else Some* [*x*,*y*])
| *conflicts* (*Var x*) (*Fun - -*) = (*Some* [*x*])
| *conflicts* (*Fun - -*) (*Var x*) = (*Some* [*x*])
| *conflicts* (*Fun f ss*) (*Fun g ts*) = (*if* (*f*,*length ss*) = (*g*,*length ts*)
then map-option concat (*those* (*map2 conflicts ss ts*))
else None)

abbreviation *Conflict-Var s t x* ≡ *conflicts s t* ≠ *None* ∧ *x* ∈ *set* (*the* (*conflicts s t*))
abbreviation *Conflict-Clash s t* ≡ *conflicts s t* = *None*

locale *pattern-completeness-context* =
fixes *S* :: *'s set* — set of sort-names
and *C* :: (*'f*,*'s*)*ssig* — sorted signature
and *m* :: *nat* — upper bound on arities of constructors
and *Cl* :: *'s* ⇒ (*'f* × *'s list*)*list* — a function to compute all constructors of given sort as list
and *inf-sort* :: *'s* ⇒ *bool* — a function to indicate whether a sort is infinite
and *ty* :: *'v itself*
begin

definition *tvars-disj-pp* :: *nat set* ⇒ (*'f*,*'v*,*'s*)*pat-problem-set* ⇒ *bool* **where**
tvars-disj-pp V p = (∀ *mp* ∈ *p*. ∀ (*ti*,*pi*) ∈ *mp*. *fst* ‘ *vars ti* ∩ *V* = {})

definition *inf-var-conflict* :: (*'f*,*'v*,*'s*)*match-problem-set* ⇒ *bool* **where**
inf-var-conflict mp = (∃ *s t x y*.
(*s*, *Var x*) ∈ *mp* ∧ (*t*, *Var x*) ∈ *mp* ∧ *Conflict-Var s t y* ∧ *inf-sort* (*snd y*))

definition *tvars-mp* :: (*'f*,*'v*,*'s*)*match-problem-set* ⇒ (*nat* × *'s*) *set* **where**
tvars-mp mp = (⋃ (*t*,*l*) ∈ *mp*. *vars t*)

definition *tvars-pp* :: (*'f*,*'v*,*'s*)*pat-problem-set* ⇒ (*nat* × *'s*) *set* **where**
tvars-pp pp = (⋃ *mp* ∈ *pp*. *tvars-mp mp*)

definition *subst-match-problem-set* :: (*'f*,*nat* × *'s*)*subst* ⇒ (*'f*,*'v*,*'s*)*match-problem-set* ⇒ (*'f*,*'v*,*'s*)*match-problem-set* **where**
subst-match-problem-set τ *pp* = *subst-left* τ ‘ *pp*

definition *subst-pat-problem-set* :: (*'f*,*nat* × *'s*)*subst* ⇒ (*'f*,*'v*,*'s*)*pat-problem-set* ⇒ (*'f*,*'v*,*'s*)*pat-problem-set* **where**
subst-pat-problem-set τ *P* = *subst-match-problem-set* τ ‘ *P*

definition τs :: $nat \Rightarrow nat \times {'s} \Rightarrow ('f,nat \times {'s})subst\ set$ **where**
$\tau s\ n\ x = \{\tau c\ n\ x\ (f,ss) \mid f\ ss.\ f : ss \to snd\ x\ in\ C\}$

The transformation rules of the paper.

The formal definition contains two deviations from the rules in the paper: first, the instantiate-rule can always be applied; and second there is an identity rule, which will simplify later refinement proofs. Both of the deviations cause non-termination.

The formal inference rules further separate those rules that deliver a bottom- or top-element from the ones that deliver a transformed problem.

inductive *mp-step* :: $('f,'v,'s)match\text{-}problem\text{-}set \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}set \Rightarrow bool$
(**infix** $\to_s$ *50*) **where**
mp-decompose: $length\ ts = length\ ls \Longrightarrow insert\ (Fun\ f\ ts,\ Fun\ f\ ls)\ mp \to_s set\ (zip\ ts\ ls) \cup mp$
| *mp-match*: $x \notin \bigcup\ (vars\ `\ snd\ `\ mp) \Longrightarrow insert\ (t,\ Var\ x)\ mp \to_s mp$
| *mp-identity*: $mp \to_s mp$

inductive *mp-fail* :: $('f,'v,'s)match\text{-}problem\text{-}set \Rightarrow bool$ **where**
mp-clash: $(f,length\ ts) \neq (g,length\ ls) \Longrightarrow mp\text{-}fail\ (insert\ (Fun\ f\ ts,\ Fun\ g\ ls)\ mp)$
| *mp-clash'*: $Conflict\text{-}Clash\ s\ t \Longrightarrow mp\text{-}fail\ (\{(s,Var\ x),(t,\ Var\ x)\} \cup mp)$

inductive *pp-step* :: $('f,'v,'s)pat\text{-}problem\text{-}set \Rightarrow ('f,'v,'s)pat\text{-}problem\text{-}set \Rightarrow bool$
(**infix** $\Rightarrow_s$ *50*) **where**
pp-simp-mp: $mp \to_s mp' \Longrightarrow insert\ mp\ pp \Rightarrow_s insert\ mp'\ pp$
| *pp-remove-mp*: $mp\text{-}fail\ mp \Longrightarrow insert\ mp\ pp \Rightarrow_s pp$

inductive *pp-success* :: $('f,'v,'s)pat\text{-}problem\text{-}set \Rightarrow bool$ **where**
$pp\text{-}success\ (insert\ \{\}\ pp)$

inductive *P-step-set* :: $('f,'v,'s)pats\text{-}problem\text{-}set \Rightarrow ('f,'v,'s)pats\text{-}problem\text{-}set \Rightarrow bool$
(**infix** $\Rrightarrow_s$ *50*) **where**
P-fail: $insert\ \{\}\ P \Rrightarrow_s bottom$
| *P-simp*: $pp \Rightarrow_s pp' \Longrightarrow insert\ pp\ P \Rrightarrow_s insert\ pp'\ P$
| *P-remove-pp*: $pp\text{-}success\ pp \Longrightarrow insert\ pp\ P \Rrightarrow_s P$
| *P-instantiate*: $tvars\text{-}disj\text{-}pp\ \{n\ ..<\ n+m\}\ pp \Longrightarrow x \in tvars\text{-}pp\ pp \Longrightarrow$
$insert\ pp\ P \Rrightarrow_s \{subst\text{-}pat\text{-}problem\text{-}set\ \tau\ pp\ |.\ \tau \in \tau s\ n\ x\} \cup P$
| *P-failure'*: $\forall\, mp \in pp.\ inf\text{-}var\text{-}conflict\ mp \Longrightarrow finite\ pp \Longrightarrow insert\ pp\ P \Rrightarrow_s \{\{\}\}$

Note that in *P-failure'* the conflicts have to be simultaneously occurring. If just some matching problem has such a conflict, then this cannot be deleted immediately!

Example-program: f(x,x) = ..., f(s(x),y) = ..., f(x,s(y)) = ... cover all cases of natural numbers, i.e., f(x1,x2), but if one would immediately delete the matching problem of the first lhs because of the resulting *inf-var-conflict* in (x1,x),(x2,x) then it is no longer complete.

3.2 Soundness of the inference rules

The empty set of variables

definition *EMPTY* :: $'v \Rightarrow 's$ *option* **where** *EMPTY* x = *None*
definition *EMPTYn* :: *nat* $\times$ $'s \Rightarrow 's$ *option* **where** *EMPTYn* x = *None*

A constructor-ground substitution for the fixed set of constructors and set of sorts. Note that variables to instantiate are represented as pairs of (number, sort).

definition *cg-subst* :: $('f,nat \times 's,'v)gsubst \Rightarrow bool$ **where**
cg-subst $\sigma = (\forall\ x.\ snd\ x \in S \longrightarrow (\sigma\ x : snd\ x\ in\ \mathcal{T}(C,EMPTY)))$

A definition of pattern completeness for pattern problems.

definition *match-complete-wrt* :: $('f,nat \times 's,'v)gsubst \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}set$ $\Rightarrow bool$ **where**
match-complete-wrt $\sigma\ mp = (\exists\ \mu.\ \forall\ (t,l) \in mp.\ t \cdot \sigma = l \cdot \mu)$

definition *pat-complete* :: $('f,'v,'s)pat\text{-}problem\text{-}set \Rightarrow bool$ **where**
pat-complete $pp = (\forall \sigma.\ cg\text{-}subst\ \sigma \longrightarrow (\exists\ mp \in pp.\ match\text{-}complete\text{-}wrt\ \sigma\ mp))$

abbreviation *pats-complete* $P \equiv \forall pp \in P.\ pat\text{-}complete\ pp$

Well-formed matching and pattern problems: all occurring variables (in left-hand sides of matching problems) have a known sort.

definition *wf-match* :: $('f,'v,'s)match\text{-}problem\text{-}set \Rightarrow bool$ **where**
wf-match $mp = (snd\ `\ tvars\text{-}mp\ mp \subseteq S)$

definition *wf-pat* :: $('f,'v,'s)pat\text{-}problem\text{-}set \Rightarrow bool$ **where**
wf-pat $pp = (\forall mp \in pp.\ wf\text{-}match\ mp)$

definition *wf-pats* :: $('f,'v,'s)pats\text{-}problem\text{-}set \Rightarrow bool$ **where**
wf-pats $P = (\forall pp \in P.\ wf\text{-}pat\ pp)$
end

lemma *type-conversion*: $t : s\ in\ \mathcal{T}(C,\emptyset) \Longrightarrow t \cdot \sigma : s\ in\ \mathcal{T}(C,\emptyset)$
proof (*induct t s rule*: *hastype-in-Term-induct*)
case (*Fun f ss σs τ*)
then show *?case* **unfolding** *eval-term.simps*
by (*smt* (*verit*, *best*) *Fun-hastype list-all2-map1 list-all2-mono*)
qed *auto*

lemma *ball-insert-un-cong*: $f\ y = Ball\ zs\ f \Longrightarrow Ball\ (insert\ y\ A)\ f = Ball\ (zs \cup A)\ f$
by *auto*

```
lemma bex-insert-cong: f y = f z ⟹ Bex (insert y A) f = Bex (insert z A) f
  by auto

lemma not-bdd-above-natD:
  assumes ¬ bdd-above (A :: nat set)
  shows ∃ x ∈ A. x > n
  using assms by (meson bdd-above.unfold linorder-le-cases order.strict-iff)

lemma list-eq-nth-eq: xs = ys ⟷ length xs = length ys ∧ (∀ i < length ys. xs !
i = ys ! i)
  using nth-equalityI by metis

lemma subt-size: p ∈ poss t ⟹ size (t |- p) ≤ size t
proof (induct p arbitrary: t)
  case (Cons i p t)
  thus ?case
  proof (cases t)
    case (Fun f ss)
    from Cons Fun have i: i < length ss and sub: t |- (i # p) = (ss ! i) |- p
      and p ∈ poss (ss ! i) by auto
    with Cons(1)[OF this(3)]
    have size (t |- (i # p)) ≤ size (ss ! i) by auto
   also have ... ≤ size t using i unfolding Fun by (simp add: termination-simp)
    finally show ?thesis .
  qed auto
qed auto

lemma conflicts-sym: rel-option (λ xs ys. set xs = set ys) (conflicts s t) (conflicts
t s) (is rel-option - (?c s t) -)
proof (induct s t rule: conflicts.induct)
  case (4 f ss g ts)
  define c where c = ?c
  show ?case
  proof (cases (f,length ss) = (g,length ts))
    case True
    hence len: length ss = length ts
      ((f, length ss) = (g, length ts)) = True
      ((g, length ts) = (f, length ss)) = True by auto
    show ?thesis using len(1) 4[OF True - refl]
      unfolding conflicts.simps len(2,3) if-True
      unfolding option.rel-map c-def[symmetric] set-concat
    proof (induct ss ts rule: list-induct2, goal-cases)
      case (2 s ss t ts)
      hence IH: rel-option (λx y. ⋃ (set ‘ set x) = ⋃ (set ‘ set y)) (those (map2
c ss ts)) (those (map2 c ts ss)) by auto
      from 2 have st: rel-option (λxs ys. set xs = set ys) (c s t) (c t s) by auto
     from IH st show ?case by (cases c s t; cases c t s; auto simp: option.rel-map)
        (simp add: option.rel-sel)
    qed simp
```

qed *auto*
qed *auto*

lemma *conflicts*: **fixes** $x :: 'v$
shows *Conflict-Clash s t* $\Longrightarrow \exists$ *p*. $p \in$ *poss s* $\wedge$ $p \in$ *poss t* $\wedge$ *is-Fun* (*s* |-*p*) $\wedge$ *is-Fun* (*t* |-*p*) $\wedge$ *root* (*s* |-*p*) $\neq$ *root* (*t* |- *p*) (**is** *?B1* $\Longrightarrow$ *?B2*)
and *Conflict-Var s t x* $\Longrightarrow$
$\exists$ *p* . $p \in$ *poss s* $\wedge$ $p \in$ *poss t* $\wedge$ *s* |-*p* $\neq$ *t* |-*p* $\wedge$ (*s* |-*p* = *Var x* $\vee$ *t* |-*p* = *Var x*) (**is** *?C1 x* $\Longrightarrow$ *?C2 x*)
and $s \neq t \Longrightarrow \exists$ *x*. *Conflict-Clash s t* $\vee$ *Conflict-Var s t x*
and *Conflict-Var s t x* $\Longrightarrow$ $x \in$ *vars s* $\cup$ *vars t*
and *conflicts s t* = *Some* [] $\longleftrightarrow$ *s* = *t* (**is** *?A*)
proof −
let *?B* = *?B1* $\longrightarrow$ *?B2*
let *?C* = λ *x*. *?C1 x* $\longrightarrow$ *?C2 x*
{
fix $x :: 'v$
have (*conflicts s t* = *Some* [] $\longrightarrow$ *s* = *t*) $\wedge$ *?B* $\wedge$ *?C x*
proof (*induction s arbitrary*: *t*)
case (*Var y t*)
thus *?case* **by** (*cases t*, *auto*)
next
case (*Fun f ss t*)
show *?case*
proof (*cases t*)
case *t*: (*Fun g ts*)
show *?thesis*
proof (*cases* (*f*,*length ss*) = (*g*,*length ts*))
case *False*
hence *res*: *conflicts* (*Fun f ss*) *t* = *None* **unfolding** *t* **by** *auto*
show *?thesis* **unfolding** *res* **unfolding** *t* **using** *False*
by (*auto intro*!: *exI*[*of* - *Nil*])
next
case *f*: *True*
let *?s* = *Fun f ss*
show *?thesis*
proof (*cases those* (*map2 conflicts ss ts*))
case *None*
hence *res*: *conflicts ?s t* = *None* **unfolding** *t* **by** *auto*
from *None*[*unfolded those-eq-None*] **obtain** *i* **where** *i*: *i* < *length ss i* < *length ts* **and**
confl: *conflicts* (*ss* ! *i*) (*ts* ! *i*) = *None*
using *f* **unfolding** *set-conv-nth set-zip* **by** *auto*
from *i* **have** *ss* ! *i* $\in$ *set ss* **by** *auto*
from *Fun.IH*[*OF this*, *of ts* ! *i*] *confl* **obtain** *p*
where *p*: $p \in$ *poss* (*ss* ! *i*) $\wedge$ $p \in$ *poss* (*ts* ! *i*) $\wedge$ *is-Fun* (*ss* ! *i* |- *p*) $\wedge$ *is-Fun* (*ts* ! *i* |- *p*) $\wedge$ *root* (*ss* ! *i* |- *p*) $\neq$ *root* (*ts* ! *i* |- *p*)
by *auto*

from p **have** p: $\exists\ p.\ p \in poss\ ?s \wedge p \in poss\ t \wedge is\text{-}Fun\ (?s\ |\text{-}\ p) \wedge is\text{-}Fun\ (t\ |\text{-}\ p) \wedge root\ (?s\ |\text{-}\ p) \neq root\ (t\ |\text{-}\ p)$
by (*intro exI*[*of* - i # p], *unfold t*, *insert i f*, *auto*)
from *p res* **show** *?thesis* **by** *auto*
next
case (*Some xss*)
hence *res*: *conflicts ?s t* = *Some* (*concat xss*) **unfolding** *t* **using** *f* **by** *auto*
from *Some* **have** *map2*: *map2 conflicts ss ts* = *map Some xss* **by** *auto*
from *arg-cong*[*OF this*, *of length*] **have** *len*: *length xss* = *length ss* **using** *f* **by** *auto*
have *rec*: $i < length\ ss \Longrightarrow conflicts\ (ss\ !\ i)\ (ts\ !\ i) = Some\ (xss\ !\ i)$ **for** i
using *arg-cong*[*OF map2*, *of* λ *xs*. *xs* ! *i*] *len f* **by** *auto*
{
assume $x \in set\ (the\ (conflicts\ ?s\ t))$
hence $x \in set\ (concat\ xss)$ **unfolding** *res* **by** *auto*
then obtain *xs* **where** *xs*: $xs \in set\ xss$ **and** *x*: $x \in set\ xs$ **by** *auto*
from *xs len* **obtain** *i* **where** *i*: $i < length\ ss$ **and** *xs*: $xs = xss\ !\ i$ **by** (*auto simp*: *set-conv-nth*)
from *i* **have** $ss\ !\ i \in set\ ss$ **by** *auto*
from *Fun.IH*[*OF this*, *of ts* ! *i*, *unfolded rec*[*OF i*, *folded xs*]] *x*
obtain *p* **where** $p \in poss\ (ss\ !\ i) \wedge p \in poss\ (ts\ !\ i) \wedge ss\ !\ i\ |\text{-}\ p \neq ts\ !\ i\ |\text{-}\ p \wedge (ss\ !\ i\ |\text{-}\ p = Var\ x \vee ts\ !\ i\ |\text{-}\ p = Var\ x)$
by *auto*
hence $\exists\ p.\ p \in poss\ ?s \wedge p \in poss\ t \wedge ?s\ |\text{-}\ p \neq t\ |\text{-}\ p \wedge (?s\ |\text{-}\ p = Var\ x \vee t\ |\text{-}\ p = Var\ x)$
by (*intro exI*[*of* - i # p], *insert i f*, *auto simp*: *t*)
}
moreover
{
assume *conflicts ?s t* = *Some* []
with *res* **have** *empty*: *concat xss* = [] **by** *auto*
{
fix *i*
assume *i*: $i < length\ ss$
from *rec*[*OF i*] **have** *conflicts* (*ss* ! *i*) (*ts* ! *i*) = *Some* (*xss* ! *i*) **.**
moreover from *empty i len* **have** *xss* ! *i* = [] **by** *auto*
ultimately have *res*: *conflicts* (*ss* ! *i*) (*ts* ! *i*) = *Some* [] **by** *simp*
from *i* **have** $ss\ !\ i \in set\ ss$ **by** *auto*
from *Fun.IH*[*OF this*, *of ts* ! *i*, *unfolded res*] **have** *ss* ! *i* = *ts* ! *i* **by** *auto*
}
with *f* **have** *?s* = *t* **unfolding** *t* **by** (*auto intro*: *nth-equalityI*)
}
ultimately show *?thesis* **unfolding** *res* **by** *auto*
qed
qed
qed *auto*

qed
} **note** *main* = *this*
from *main* **show** *B*: *?B1* $\Longrightarrow$ *?B2* **and** *C*: *?C1 x* $\Longrightarrow$ *?C2 x* **by** *blast+*
show *?A*
proof
assume *s* = *t*
with *B* **have** *conflicts s t* $\neq$ *None* **by** *blast*
then obtain *xs* **where** *res*: *conflicts s t* = *Some xs* **by** *auto*
show *conflicts s t* = *Some* []
proof (*cases xs*)
case *Nil*
thus *?thesis* **using** *res* **by** *auto*
next
case (*Cons x xs*)
with *main*[*of x*] *res* ‹*s* = *t*› **show** *?thesis* **by** *auto*
qed
qed (*insert main*, *blast*)
{
assume *diff*: *s* $\neq$ *t*
show $\exists$ *x*. *Conflict-Clash s t* $\vee$ *Conflict-Var s t x*
proof (*cases conflicts s t*)
case (*Some xs*)
with ‹*?A*› *diff* **obtain** *x* **where** *x* $\in$ *set xs* **by** (*cases xs*, *auto*)
thus *?thesis* **unfolding** *Some* **by** *auto*
qed *auto*
}
assume *Conflict-Var s t x*
with *C* **obtain** *p* **where** *p* $\in$ *poss s p* $\in$ *poss t* (*s* |- *p* = *Var x* $\vee$ *t* |- *p* = *Var x*)
by *blast*
thus *x* $\in$ *vars s* $\cup$ *vars t*
by (*metis UnCI subt-at-imp-supteq′ subteq-Var-imp-in-vars-term*)
qed

declare *conflicts.simps*[*simp del*]

lemma *conflicts-refl*[*simp*]: *conflicts t t* = *Some* []
using *conflicts*(*5*)[*of t t*] **by** *auto*

For proving partial correctness we need further properties of the fixed parameters: We assume that *m* is sufficiently large and that there exists some constructor ground terms. Moreover *inf-sort* really computes whether a sort has terms of arbitrary size. Further all symbols in *C* must have sorts of *S*. Finally, *Cl* should precisely compute the constructors of a sort.

locale *pattern-completeness-context-with-assms* = *pattern-completeness-context S C m Cl inf-sort ty*
for *S* **and** *C* :: (*′f*,*′s*)*ssig*
and *m Cl inf-sort*
and *ty* :: *′v itself* +

assumes *sorts-non-empty*: $\bigwedge s.\ s \in S \Longrightarrow \exists\ t.\ t : s\ in\ \mathcal{T}(C,\ EMPTY)$
and *C-sub-S*: $\bigwedge f\ ss\ s.\ f : ss \to s\ in\ C \Longrightarrow insert\ s\ (set\ ss) \subseteq S$
and *m*: $\bigwedge f\ ss\ s.\ f : ss \to s\ in\ C \Longrightarrow length\ ss \leq m$
and *inf-sort-def*: $s \in S \Longrightarrow inf\text{-}sort\ s\ = (\neg\ bdd\text{-}above\ (size\ `\ \{t\ .\ t : s\ in\ \mathcal{T}(C,EMPTYn)\}))$
and *Cl*: $\bigwedge s.\ set\ (Cl\ s) = \{(f,ss).\ f : ss \to s\ in\ C\}$
and *Cl-len*: $\bigwedge \sigma.\ Ball\ (length\ `\ snd\ `\ set\ (Cl\ \sigma))\ (\lambda\ a.\ a \leq m)$
begin

lemmas *subst-defs-set* =
subst-pat-problem-set-def
subst-match-problem-set-def

Preservation of well-formedness

lemma *mp-step-wf*: $mp \to_s mp' \Longrightarrow wf\text{-}match\ mp \Longrightarrow wf\text{-}match\ mp'$
unfolding *wf-match-def tvars-mp-def*
proof (*induct mp mp′ rule*: *mp-step.induct*)
case (*mp-decompose f ts ls mp*)
then show *?case* **by** (*auto dest!*: *set-zip-leftD*)
qed *auto*

lemma *pp-step-wf*: $pp \Rightarrow_s pp' \Longrightarrow wf\text{-}pat\ pp \Longrightarrow wf\text{-}pat\ pp'$
unfolding *wf-pat-def*
proof (*induct pp pp′ rule*: *pp-step.induct*)
case (*pp-simp-mp mp mp′ pp*)
then show *?case* **using** *mp-step-wf*[*of mp mp′*] **by** *auto*
qed *auto*

theorem *P-step-set-wf*: $P \Rrightarrow_s P' \Longrightarrow wf\text{-}pats\ P \Longrightarrow wf\text{-}pats\ P'$
unfolding *wf-pats-def*
proof (*induct P P′ rule*: *P-step-set.induct*)
case (*P-simp pp pp′ P*)
then show *?case* **using** *pp-step-wf*[*of pp pp′*] **by** *auto*
next
case ∗: (*P-instantiate n p x P*)
let *?s* = *snd x*
from ∗ **have** *sS*: $?s \in S$ **and** *p*: *wf-pat p* **unfolding** *wf-pat-def wf-match-def tvars-pp-def* **by** *auto*
{
fix τ
assume *tau*: $\tau \in \tau s\ n\ x$
from *tau*[*unfolded* τs*-def* τc*-def*, *simplified*]
obtain *f sorts* **where** *f*: $f : sorts \to snd\ x\ in\ C$ **and** τ: $\tau = subst\ x\ (Fun\ f\ (map\ Var\ (zip\ [n..<n + length\ sorts]\ sorts)))$ **by** *auto*
let *?i* = *length sorts*
let *?xs* = *zip* [*n*..<*n* + *length sorts*] *sorts*
from *C-sub-S*[*OF f*] **have** *sS*: $?s \in S$ **and** *xs*: $snd\ `\ set\ ?xs \subseteq S$
unfolding *set-conv-nth set-zip* **by** *auto*

{
fix *mp y*
assume *mp*: $mp \in p$ **and** $y \in$ *tvars-mp* (*subst-left* τ ‘ *mp*)
then obtain *s t* **where** *y*: $y \in$ *vars* $(s \cdot \tau)$ **and** *st*: $(s,t) \in mp$
unfolding *tvars-mp-def subst-left-def* **by** *auto*
from *y* **have** $y \in$ *vars s* $\cup$ *set ?xs* **unfolding** *vars-term-subst* τ
by (*auto simp*: *subst-def split*: *if-splits*)
hence *snd y* $\in$ *snd* ‘ *vars s* $\cup$ *snd* ‘ *set ?xs* **by** *auto*
also have ... $\subseteq$ *snd* ‘ *vars s* $\cup$ *S* **using** *xs* **by** *auto*
also have ... $\subseteq$ *S* **using** *p mp st*
unfolding *wf-pat-def wf-match-def tvars-mp-def* **by** *force*
finally have *snd y* $\in$ *S* **.**
}
hence *wf-pat* (*subst-pat-problem-set* τ *p*)
unfolding *wf-pat-def wf-match-def subst-defs-set* **by** *auto*
}
with $*$ **show** *?case* **by** *auto*
qed (*auto simp*: *wf-pat-def*)

Soundness requires some preparations

lemma *cg-exists*: $\exists$ σg. *cg-subst* σg
proof
show *cg-subst* (λ *x*. *SOME t*. *t* : *snd x in* $\mathcal{T}$(*C*, *EMPTY*))
unfolding *cg-subst-def*
proof (*intro allI impI*, *goal-cases*)
case (*1 x*)
from *someI-ex*[*OF sorts-non-empty*[*OF 1*]] **show** *?case* **by** *simp*
qed
qed

definition σg :: (*'f*,*nat* $\times$ *'s*,*'v*)*gsubst* **where** σg = (*SOME* σ. *cg-subst* σ)

lemma σg: *cg-subst* σg **unfolding** σg*-def* **using** *cg-exists* **by** (*metis someI-ex*)

lemma *pat-complete-empty*[*simp*]: *pat-complete* {} = *False*
unfolding *pat-complete-def* **using** σg **by** *auto*

lemma *inf-var-conflictD*: **assumes** *inf-var-conflict mp*
shows $\exists$ *p s t x y*.
$(s, Var\ x) \in mp \wedge (t, Var\ x) \in mp \wedge s$ |-*p* = *Var y* $\wedge$ *s* |- *p* $\neq$ *t* |-*p* $\wedge$ *p* $\in$ *poss s* $\wedge$ *p* $\in$ *poss t* $\wedge$ *inf-sort* (*snd y*)
proof −
from *assms*[*unfolded inf-var-conflict-def*]
obtain *s t x y* **where** $(s,\ Var\ x) \in mp \wedge (t,\ Var\ x) \in mp$ **and** *conf*: *Conflict-Var s t y* **and** *y*: *inf-sort* (*snd y*) **by** *blast*
with *conflicts*(*2*)[*OF conf*] **show** *?thesis* **by** *metis*
qed

lemma *cg-term-vars*: *t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*) $\Longrightarrow$ *vars t* = {}

proof (*induct t s rule*: *hastype-in-Term-induct*)
  case (*Var v σ*)
  then show *?case* **by** (*auto simp*: *EMPTYn-def*)
next
  case (*Fun f ss σs τ*)
  then show *?case* **unfolding** *term.simps* **unfolding** *set-conv-nth list-all2-conv-all-nth*
by *auto*
qed

lemma *type-conversion1*: $t : s\ in\ \mathcal{T}(C,EMPTYn) \Longrightarrow t \cdot \sigma' : s\ in\ \mathcal{T}(C,EMPTY)$

  unfolding *EMPTYn-def EMPTY-def* **by** (*rule type-conversion*)

lemma *type-conversion2*: $t : s\ in\ \mathcal{T}(C,EMPTY) \Longrightarrow t \cdot \sigma' : s\ in\ \mathcal{T}(C,EMPTYn)$

  unfolding *EMPTYn-def EMPTY-def* **by** (*rule type-conversion*)

lemma *term-of-sort*: **assumes** $s \in S$
  shows $\exists\ t.\ t : s\ in\ \mathcal{T}(C,EMPTYn)$
proof −
  from *σg*[*unfolded cg-subst-def*] *assms*
  have $\exists\ t.\ t : s\ in\ \mathcal{T}(C,EMPTY)$ **by** *force*
  with *type-conversion2*[*of - s*]
  show *?thesis* **by** *auto*
qed

Main partial correctness theorems on well-formed problems: the transformation rules do not change the semantics of a problem

lemma *mp-step-pcorrect*: $mp \rightarrow_s mp' \Longrightarrow$ *match-complete-wrt* σ *mp* = *match-complete-wrt* σ mp'
proof (*induct mp mp' rule*: *mp-step.induct*)
  case ∗: (*mp-decompose f ts ls mp*)
  show *?case* **unfolding** *match-complete-wrt-def*
    apply (*rule ex-cong1*)
    apply (*rule ball-insert-un-cong*)
    apply (*unfold split*) **using** ∗ **by** (*auto simp add*: *set-zip list-eq-nth-eq*)
next
  case ∗: (*mp-match x mp t*)
  show *?case* **unfolding** *match-complete-wrt-def*
  proof
    assume $\exists\mu.\ \forall(ti,\ li)\in mp.\ ti \cdot \sigma = li \cdot \mu$
    then obtain μ **where** *eq*: $\bigwedge ti\ li.\ (ti,\ li)\in mp \Longrightarrow\ ti \cdot \sigma = li \cdot \mu$ **by** *auto*
    let $?\mu = \mu(x := t \cdot \sigma)$
    have $(ti,\ li) \in mp \Longrightarrow ti \cdot \sigma = li \cdot\ ?\mu$ **for** *ti li* **using** ∗ *eq*[*of ti li*]
      by (*auto intro*!: *term-subst-eq*)
    thus $\exists\mu.\ \forall(ti,\ li)\in insert\ (t,\ Var\ x)\ mp.\ ti \cdot \sigma = li \cdot \mu$ **by** (*intro exI*[*of - ?μ*], *auto*)
  qed *auto*
qed *auto*

lemma *mp-fail-pcorrect*: *mp-fail mp* $\Longrightarrow$ $\neg$ *match-complete-wrt* σ *mp*
proof (*induct mp rule*: *mp-fail.induct*)
  case $*$: (*mp-clash f ts g ls mp*)
  {
    assume *length ts* $\neq$ *length ls*
    hence (*map* ($\lambda t.\ t \cdot \mu$) *ls* = *map* ($\lambda t.\ t \cdot \sigma$) *ts*) = *False* **for** σ :: ($'f$,*nat* $\times$ $'s,'a$)*gsubst* **and** μ
      by (*metis length-map*)
  } **note** *len* = *this*
  from $*$ **show** *?case* **unfolding** *match-complete-wrt-def*
    by (*auto simp*: *len*)
next
  case $*$: (*mp-clash' s t x mp*)
  from *conflicts*(*1*)[*OF* $*$(*1*)]
  obtain *po* **where** $*$: *po* $\in$ *poss s po* $\in$ *poss t is-Fun* (*s* |- *po*) *is-Fun* (*t* |- *po*) *root* (*s* |- *po*) $\neq$ *root* (*t* |- *po*)
    by *auto*
  show *?case*
  proof
    assume *match-complete-wrt* σ ({(*s*, *Var x*), (*t*, *Var x*)} $\cup$ *mp*)
    from *this*[*unfolded match-complete-wrt-def*]
    have $s \cdot \sigma = t \cdot \sigma$ **by** *auto*
    hence *root* ($s \cdot \sigma$ |-*po*) = *root* ($t \cdot \sigma$ |-*po*) **by** *auto*
    also have *root* ($s \cdot \sigma$ |-*po*) = *root* (*s* |-*po* $\cdot\ \sigma$) **using** $*$ **by** *auto*
    also have ... = *root* (*s* |-*po*) **using** $*$ **by** (*cases s* |- *po*, *auto*)
    also have *root* ($t \cdot \sigma$ |-*po*) = *root* (*t* |-*po* $\cdot\ \sigma$) **using** $*$ **by** (*cases t* |- *po*, *auto*)
    also have ... = *root* (*t* |-*po*) **using** $*$ **by** (*cases t* |- *po*, *auto*)
    finally show *False* **using** $*$ **by** *auto*
  qed
qed

lemma *pp-step-pcorrect*: *pp* $\Rightarrow_s$ *pp'* $\Longrightarrow$ *pat-complete pp* = *pat-complete pp'*
proof (*induct pp pp' rule*: *pp-step.induct*)
  case (*pp-simp-mp mp mp' pp*)
  then show *?case* **using** *mp-step-pcorrect*[*of mp mp'*] **unfolding** *pat-complete-def* **by** *auto*
next
  case (*pp-remove-mp mp pp*)
  then show *?case* **using** *mp-fail-pcorrect*[*of mp*] **unfolding** *pat-complete-def* **by** *auto*
qed

lemma *pp-success-pcorrect*: *pp-success pp* $\Longrightarrow$ *pat-complete pp*
  by (*induct pp rule*: *pp-success.induct*, *auto simp*: *pat-complete-def match-complete-wrt-def*)

theorem *P-step-set-pcorrect*: *P* $\Rrightarrow_s$ *P'* $\Longrightarrow$ *wf-pats P* $\Longrightarrow$

pats-complete $P \longleftrightarrow$ *pats-complete* P'
proof (*induct* P P' *rule*: *P-step-set.induct*)
  case (*P-fail* P)
  then show *?case* **by** (*auto simp*: *pat-complete-def*)
next
  case (*P-simp pp pp' P*)
  then show *?case* **using** *pp-step-pcorrect*[*of pp pp'*] **by** *auto*
next
  case (*P-remove-pp pp P*)
  then show *?case* **using** *pp-success-pcorrect*[*of pp*] **by** *auto*
next
  case $*$: (*P-instantiate n pp x P*)
  note *def* = *pat-complete-def*[*unfolded match-complete-wrt-def*]
  show *?case*
  proof (*rule ball-insert-un-cong*, *standard*)
    assume *complete*: *pats-complete* {*subst-pat-problem-set* τ *pp* |. $\tau \in \tau s$ *n x*}
    show *pat-complete pp* **unfolding** *def*
    proof (*intro allI impI*)
      fix σ :: (*'f*,*nat* $\times$ *'s*,*'v*)*gsubst*

      from $*$ **have** *wf-pat pp* **unfolding** *wf-pats-def* **by** *auto*
      with $*(2)$ **have** *x*: *snd x* $\in$ *S* **unfolding** *tvars-pp-def tvars-mp-def wf-pat-def wf-match-def* **by** *force*

      assume *cg*: *cg-subst* σ
      from *this*[*unfolded cg-subst-def*] *x*
      have σ *x* : *snd x in* $\mathcal{T}$(*C*,*EMPTY*) **by** *blast*
      then obtain *f ts* σs **where** *f*: *f* : $\sigma s \to$ *snd x in C*
        and *args*: *ts* $:_l$ σs *in* $\mathcal{T}$(*C*,*EMPTY*)
        and σx: σ *x* = *Fun f ts*
        by (*induct*, *auto simp*: *EMPTY-def*)
      from *f* **have** *f*: *f* : $\sigma s \to$ *snd x in C*
        by (*meson hastype-in-ssig-def*)
      let *?l* = *length ts*
      from *args* **have** *len*: *length* σs = *?l*
        by (*simp add*: *list-all2-lengthD*)
      have *l*: *?l* $\leq$ *m* **using** *m*[*OF f*] *len* **by** *auto*
      define σ' **where** σ' = (λ *ys*. *let y* = *fst ys in if n* $\leq$ *y* $\wedge$ *y* < *n* + *?l* $\wedge$ σs ! (*y* − *n*) = *snd ys then ts* ! (*y* − *n*) *else* σ *ys*)
      have *cg*: *cg-subst* σ' **unfolding** *cg-subst-def*
      proof (*intro allI impI*)
        fix *ys* :: *nat* $\times$ *'s*
        assume *ysS*: *snd ys* $\in$ *S*
        show σ' *ys* : *snd ys in* $\mathcal{T}$(*C*,*EMPTY*)
        proof (*cases* σ' *ys* = σ *ys*)
          case *True*
          thus *?thesis* **using** *cg ysS* **unfolding** *cg-subst-def* **by** *metis*
        next
          case *False*

obtain *y s* **where** *ys*: $ys = (y,s)$ **by** *force*
with *False* **have** *y*: $y - n < ?l\ n \le y\ y < n + ?l$ **and** *arg*: $\sigma s\ !\ (y - n) = s$ **and** σ': $\sigma'\ ys = ts\ !\ (y - n)$
unfolding σ'*-def Let-def* **by** (*auto split*: *if-splits*)
show *?thesis* **unfolding** σ' **unfolding** *ys snd-conv arg*[*symmetric*] **using** *y*(*1*) *len args*
by (*metis list-all2-nthD*)
qed
qed
define τ **where** $\tau = subst\ x\ (Fun\ f\ (map\ Var\ (zip\ [n..<n + ?l]\ \sigma s)))$
from *f* **have** $\tau \in \tau s\ n\ x$ **unfolding** τs*-def* τ*-def* τc*-def* **using** *len*[*symmetric*] **by** *auto*
hence *pat-complete* (*subst-pat-problem-set* τ *pp*) **using** *complete* **by** *auto*
from *this*[*unfolded def*, *rule-format*, *OF cg*]
obtain *tl* μ **where** *tl*: $tl \in subst\text{-}pat\text{-}problem\text{-}set\ \tau\ pp$
and *match*: $\bigwedge ti\ li.\ (ti, li) \in tl \Longrightarrow ti \cdot \sigma' = li \cdot \mu$ **by** *force*
from *tl*[*unfolded subst-defs-set subst-left-def set-map*]
obtain *tl'* **where** *tl'*: $tl' \in pp$ **and** *tl*: $tl = \{(t' \cdot \tau, l)\ |.\ (t',l) \in tl'\}$ **by** *auto*
show $\exists tl\in pp.\ \exists \mu.\ \forall (ti, li)\in tl.\ ti \cdot \sigma = li \cdot \mu$
proof (*intro bexI*[*OF - tl'*] *exI*[*of - μ*], *clarify*)
fix *ti li*
assume *tli*: $(ti, li) \in tl'$
hence *tlit*: $(ti \cdot \tau, li) \in tl$ **unfolding** *tl* **by** *force*
from *match*[*OF this*] **have** *match*: $ti \cdot \tau \cdot \sigma' = li \cdot \mu$ **by** *auto*
from $*(1)$[*unfolded tvars-disj-pp-def*, *rule-format*, *OF tl' tli*]
have *vti*: *fst* ' *vars-term ti* $\cap \{n..<n + m\} = \{\}$ **by** *auto*
have $ti \cdot \sigma = ti \cdot (\tau \circ_s \sigma')$
proof (*rule term-subst-eq*, *unfold subst-compose-def*)
fix *y*
assume $y \in vars\text{-}term\ ti$
with *vti* **have** *y*: $fst\ y \notin \{n..<n + m\}$ **by** *auto*
show $\sigma\ y = \tau\ y \cdot \sigma'$
proof (*cases* $y = x$)
case *False*
hence $\tau\ y \cdot \sigma' = \sigma'\ y$ **unfolding** τ*-def subst-def* **by** *auto*
also have $\ldots = \sigma\ y$
unfolding σ'*-def* **using** *y l* **by** *auto*
finally show *?thesis* **by** *simp*
next
case *True*
show *?thesis* **unfolding** *True* τ*-def subst-simps* σx *eval-term.simps map-map o-def term.simps*
by (*intro conjI refl nth-equalityI*, *auto simp*: *len* σ'*-def*)
qed
qed
also have $\ldots = li \cdot \mu$ **using** *match* **by** *simp*
finally show $ti \cdot \sigma = li \cdot \mu$ **by** *blast*
qed
qed

next
assume *complete*: *pat-complete pp*
{
fix τ
assume $\tau \in \tau s\ n\ x$
from *this*[*unfolded* τ*s-def* τ*c-def*, *simplified*]
obtain *f sorts* **where** *f*: *f* : *sorts* $\rightarrow$ *snd x in C* **and** τ: τ = *subst x* (*Fun f* (*map Var* (*zip* [*n*..<*n* + *length sorts*] *sorts*))) **by** *auto*
let *?i* = *length sorts*
let *?xs* = *zip* [*n*..<*n* + *length sorts*] *sorts*
have *i*: *?i* $\leq$ *m* **by** (*rule m*[*OF f*])
have *pat-complete* (*subst-pat-problem-set* τ *pp*) **unfolding** *def*
proof (*intro allI impI*)
fix σ :: ('*f*,*nat* $\times$ '*s*,'*v*)*gsubst*
assume *cg*: *cg-subst* σ
define σ' **where** $\sigma' = \sigma(x :=$ *Fun f* (*map* σ *?xs*))
from *C-sub-S*[*OF f*] **have** *sortsS*: *set sorts* $\subseteq$ *S* **by** *auto*
from *f* **have** *f*: *f* : *sorts* $\rightarrow$ *snd x in C* **by** (*simp add*: *hastype-in-ssig-def*)
hence *Fun f* (*map* σ *?xs*) : *snd x in* $\mathcal{T}$(*C*,*EMPTY*)
proof (*rule Fun-hastypeI*)
show *map* σ *?xs* $:_l$ *sorts in* $\mathcal{T}$(*C*,*EMPTY*)
using *cg*[*unfolded cg-subst-def*, *rule-format*, *OF set-mp*[*OF sortsS*]]
by (*smt* (*verit*) *add-diff-cancel-left'* *length-map length-upt length-zip list-all2-conv-all-nth min.idem nth-map nth-mem nth-zip prod.sel*(*2*))
qed
hence *cg*: *cg-subst* σ' **using** *cg f* **unfolding** *cg-subst-def* σ'*-def* **by** *auto*
from *complete*[*unfolded def*, *rule-format*, *OF this*]
obtain *tl* μ **where** *tl*: *tl* $\in$ *pp* **and** *tli*: $\bigwedge$ *ti li*. (*ti*, *li*)$\in$ *tl* $\Longrightarrow$ *ti* $\cdot$ σ' = *li* $\cdot$ μ **by** *force*
from *tl* **have** *tlm*: {(*t* $\cdot$ τ, *l*) |. (*t*,*l*) $\in$ *tl*} $\in$ *subst-pat-problem-set* τ *pp*
unfolding *subst-defs-set subst-left-def* **by** *auto*
{
fix *ti li*
assume *mem*: (*ti*, *li*) $\in$ *tl*
from $*$[*unfolded tvars-disj-pp-def*] *tl mem* **have** *vti*: *fst* ‘ *vars-term ti* $\cap$ {*n*..<*n* + *m*} = {} **by** *force*
from *tli*[*OF mem*] **have** *li* $\cdot$ μ = *ti* $\cdot$ σ' **by** *auto*
also have ... = *ti* $\cdot$ ($\tau \circ_s \sigma$)
proof (*intro term-subst-eq*, *unfold subst-compose-def*)
fix *y*
assume *y* $\in$ *vars-term ti*
with *vti* **have** *y*: *fst y* $\notin$ {*n*..<*n* + *m*} **by** *auto*
show σ' *y* = τ *y* $\cdot$ σ
proof (*cases y* = *x*)
case *False*
hence τ *y* $\cdot$ σ = σ *y* **unfolding** τ *subst-def* **by** *auto*
also have ... = σ' *y*
unfolding σ'*-def* **using** *False* **by** *auto*
finally show *?thesis* **by** *simp*

next
case *True*
show *?thesis* **unfolding** *True* τ
by (*simp add*: *o-def* σ'*-def*)
qed
qed
finally have $ti \cdot \tau \cdot \sigma = li \cdot \mu$ **by** *auto*
}
thus $\exists\, tl \in$ *subst-pat-problem-set* τ *pp*. $\exists\, \mu.\ \forall\, (ti,\ li) \in tl.\ ti \cdot \sigma = li \cdot \mu$
by (*intro bexI*[*OF - tlm*], *auto*)
qed
}
thus *pats-complete* {*subst-pat-problem-set* τ *pp* |. $\tau \in \tau s\ n\ x$} **by** *auto*
qed
next
case $*$: (*P-failure'* *pp P*)
{
assume *pp*: *pat-complete pp*
with $*$(*3*) **have** *wf*: *wf-pat pp* **by** (*auto simp*: *wf-pats-def*)
define *confl'* :: ('*f*, *nat* $\times$ '*s*) *term* $\Rightarrow$ ('*f*, *nat* $\times$ '*s*)*term* $\Rightarrow$ *nat* $\times$ '*s* $\Rightarrow$ *bool*
where *confl'* = (λ *sp tp y*.
sp = *Var y* $\wedge$ *inf-sort* (*snd y*) $\wedge$ *sp* $\neq$ *tp*)
define *P1* **where** *P1* = (λ *mp s t x y p*. *mp* $\in$ *pp* $\longrightarrow$ (*s*, *Var x*) $\in$ *mp* $\wedge$ (*t*, *Var x*) $\in$ *mp* $\wedge$ *p* $\in$ *poss s* $\wedge$ *p* $\in$ *poss t* $\wedge$ *confl'* (*s* |- *p*) (*t* |- *p*) *y*)
{
fix *mp*
assume *mp* $\in$ *pp*
hence *inf-var-conflict mp* **using** $*$ **by** *auto*
from *inf-var-conflictD*[*OF this*]
have $\exists$ *s t x y p*. *P1 mp s t x y p* **unfolding** *P1-def confl'-def* **by** *force*
}
hence $\forall$ *mp*. $\exists$ *s t x y p*. *P1 mp s t x y p* **unfolding** *P1-def* **by** *blast*
from *choice*[*OF this*] **obtain** *s* **where** $\forall$ *mp*. $\exists$ *t x y p*. *P1 mp* (*s mp*) *t x y p* **by** *blast*
from *choice*[*OF this*] **obtain** *t* **where** $\forall$ *mp*. $\exists$ *x y p*. *P1 mp* (*s mp*) (*t mp*) *x y p* **by** *blast*
from *choice*[*OF this*] **obtain** *x* **where** $\forall$ *mp*. $\exists$ *y p*. *P1 mp* (*s mp*) (*t mp*) (*x mp*) *y p* **by** *blast*
from *choice*[*OF this*] **obtain** *y* **where** $\forall$ *mp*. $\exists$ *p*. *P1 mp* (*s mp*) (*t mp*) (*x mp*) (*y mp*) *p* **by** *blast*
from *choice*[*OF this*] **obtain** *p* **where** $\forall$ *mp*. *P1 mp* (*s mp*) (*t mp*) (*x mp*) (*y mp*) (*p mp*) **by** *blast*
note *P1* = *this*[*unfolded P1-def*, *rule-format*]
from $*$(*2*) **have** *finite* (*y* ‘ *pp*) **by** *blast*
from *ex-bij-betw-finite-nat*[*OF this*] **obtain** *index* **and** *n* :: *nat* **where**
bij: *bij-betw index* (*y* ‘ *pp*) {..<*n*}
by (*auto simp add*: *atLeast0LessThan*)
define *var-ind* :: *nat* $\Rightarrow$ *nat* $\times$ '*s* $\Rightarrow$ *bool* **where**
var-ind i x = (*x* $\in$ *y* ‘ *pp* $\wedge$ *index x* $\in$ {..<*n*} $-$ {..<*i*}) **for** *i x*

have [*simp*]: *var-ind n x* = *False* **for** *x*
unfolding *var-ind-def* **by** *auto*
define *cg-subst-ind* :: *nat* ⇒ ('*f*,*nat* × '*s*)*subst* ⇒ *bool* **where**
cg-subst-ind i σ = (∀ *x*. (*var-ind i x* ⟶ *σ x* = *Var x*)
∧ (¬ *var-ind i x* ⟶ (*vars-term* (*σ x*) = {} ∧ (*snd x* ∈ *S* ⟶ *σ x* : *snd x in* $\mathcal{T}$(*C*,*EMPTYn*))))) **for** *i σ*
define *confl* :: *nat* ⇒ ('*f*, *nat* × '*s*) *term* ⇒ ('*f*, *nat* × '*s*)*term* ⇒ *bool* **where**
confl = (λ *i sp tp*.
(*case* (*sp*,*tp*) *of* (*Var x*, *Var y*) ⇒ *x* ≠ *y* ∧ *var-ind i x* ∧ *var-ind i y*
| (*Var x*, *Fun* - -) ⇒ *var-ind i x*
| (*Fun* - -, *Var x*) ⇒ *var-ind i x*
| (*Fun f ss*, *Fun g ts*) ⇒ (*f*,*length ss*) ≠ (*g*,*length ts*)))
have *confl-n*: *confl n s t* ⟹ ∃ *f g ss ts*. *s* = *Fun f ss* ∧ *t* = *Fun g ts* ∧ (*f*,*length ss*) ≠ (*g*,*length ts*) **for** *s t*
by (*cases s*; *cases t*; *auto simp*: *confl-def*)
{
fix *i*
assume *i* ≤ *n*
hence ∃ *σ*. *cg-subst-ind i σ* ∧ (∀ *mp* ∈ *pp*. ∃ *p*. *p* ∈ *poss* (*s mp* · *σ*) ∧ *p* ∈ *poss* (*t mp* · *σ*) ∧ *confl i* (*s mp* · *σ* |- *p*) (*t mp* · *σ* |- *p*))
proof (*induction i*)
case *0*
define *σ* **where** *σ x* = (*if var-ind 0 x then Var x else if snd x* ∈ *S then map-vars undefined* (*σg x*) *else Fun undefined* []) **for** *x*
{
fix *x* :: *nat* × '*s*
define *t* **where** *t* = *σg x*
define *s* **where** *s* = *snd x*
assume *snd x* ∈ *S*
hence *σg x* : *snd x in* $\mathcal{T}$(*C*,*EMPTY*) **using** *σg* **unfolding** *cg-subst-def* **by** *blast*
hence *map-vars undefined* (*σg x*) : *snd x in* $\mathcal{T}$(*C*,*EMPTYn*) (**is** *?m* : - *in* -)
unfolding *t-def*[*symmetric*] *s-def*[*symmetric*]
proof (*induct t s rule*: *hastype-in-Term-induct*)
case (*Var v σ*)
then show *?case* **by** (*auto simp*: *EMPTY-def*)
next
case (*Fun f ss σs τ*)
then show *?case* **unfolding** *term.simps*
by (*smt* (*verit*, *best*) *Fun-hastype list-all2-map1 list-all2-mono*)
qed
}
from *this cg-term-vars*[*OF this*] **have** *σ*: *cg-subst-ind 0 σ* **unfolding** *cg-subst-ind-def σ-def* **by** *auto*
show *?case*
proof (*rule exI*, *rule conjI*[*OF σ*], *intro ballI exI conjI*)
fix *mp*
assume *mp*: *mp* ∈ *pp*

```
note P1 = P1[OF this]
from mp have mem: y mp ∈ y ‘ pp by auto
with bij have y: index (y mp) ∈ {..<n} by (metis bij-betw-apply)
hence y0: var-ind 0 (y mp) using mem unfolding var-ind-def by auto
show p mp ∈ poss (s mp · σ) using P1 by auto
show p mp ∈ poss (t mp · σ) using P1 by auto
let ?t = t mp |- p mp
define c where c = confl 0 (s mp · σ |- p mp) (t mp · σ |- p mp)
have c = confl 0 (s mp |- p mp · σ) (?t · σ)
  using P1 unfolding c-def by auto
also have s: s mp |- p mp = Var (y mp) using P1 unfolding confl′-def
by auto
also have . . . · σ = Var (y mp) using y0 unfolding σ-def by auto
also have confl 0 (Var (y mp)) (?t · σ)
proof (cases ?t · σ)
  case Fun
  thus ?thesis using y0 unfolding confl-def by auto
next
  case (Var z)
  then obtain u where t: ?t = Var u and ssig: σ u = Var z
    by (cases ?t, auto)
  from P1[unfolded s] have confl′ (Var (y mp)) ?t (y mp) by auto
  from this[unfolded confl′-def t] have uy: y mp ≠ u by auto
  show ?thesis
  proof (cases var-ind 0 u)
    case True
    with y0 uy show ?thesis unfolding t σ-def confl-def by auto
  next
    case False
     with ssig[unfolded σ-def] have uS: snd u ∈ S and contra: map-vars
undefined (σg u) = Var z
      by (auto split: if-splits)
    from σg[unfolded cg-subst-def, rule-format, OF uS] contra
    have False by (cases σg u, auto simp: EMPTY-def)
    thus ?thesis ..
  qed
qed
 finally show confl 0 (s mp · σ |- p mp) (t mp · σ |- p mp) unfolding
c-def .
qed
next
case (Suc i)
 then obtain σ where σ: cg-subst-ind i σ and confl: (∀ mp∈pp. ∃ p. p ∈
poss (s mp · σ) ∧ p ∈ poss (t mp · σ) ∧ confl i (s mp · σ |- p) (t mp · σ |- p))
  by auto
from Suc have i ∈ {..< n} and i: i < n by auto
with bij obtain z where z: z ∈ y ‘ pp index z = i unfolding bij-betw-def
by (metis imageE)
{
```

from z **obtain** mp **where** $mp \in pp$ **and** $index\ (y\ mp) = i$ **and** $z = y\ mp$ **by** $auto$
with $P1[OF\ this(1),\ unfolded\ confl'\text{-}def]$ **have** inf: $inf\text{-}sort\ (snd\ z)$
and $*$: $p\ mp \in poss\ (s\ mp)\ s\ mp\ |\text{-}\ p\ mp = Var\ z\ (s\ mp,\ Var\ (x\ mp)) \in mp$
by $auto$
from $*(1,2)$ **have** $z \in vars\ (s\ mp)$ **using** $vars\text{-}term\text{-}subt\text{-}at$ **by** $fastforce$
with $*(3)$ **have** $z \in tvars\text{-}mp\ mp$ **unfolding** $tvars\text{-}mp\text{-}def$ **by** $force$
with $\langle mp \in pp\rangle\ wf$ **have** $snd\ z \in S$ **unfolding** $wf\text{-}pat\text{-}def\ wf\text{-}match\text{-}def$ **by** $auto$
from $not\text{-}bdd\text{-}above\text{-}natD[OF\ inf[unfolded\ inf\text{-}sort\text{-}def[OF\ this]]]\ term\text{-}of\text{-}sort[OF\ this]$
have $\bigwedge\ n.\ \exists\ t.\ t : snd\ z\ in\ \mathcal{T}(C,EMPTYn) \wedge n < size\ t$ **by** $auto$
} **note** $z\text{-}inf = this$

define $all\text{-}st$ **where** $all\text{-}st = (\lambda\ mp.\ s\ mp \cdot \sigma)\ `\ pp \cup (\lambda\ mp.\ t\ mp \cdot \sigma)\ `\ pp$
have $fin\text{-}all\text{-}st$: $finite\ all\text{-}st$ **unfolding** $all\text{-}st\text{-}def$ **using** $*(2)$ **by** $simp$
define $d :: nat$ **where** $d = Suc\ (Max\ (size\ `\ all\text{-}st))$
from $z\text{-}inf[of\ d]$
obtain u **where** u: $u : snd\ z\ in\ \mathcal{T}(C,EMPTYn)$ **and** du: $d \le size\ u$ **by** $auto$
have $vars\text{-}u$: $vars\ u = \{\}$ **by** $(rule\ cg\text{-}term\text{-}vars[OF\ u])$

define σ' **where** $\sigma'\ x = (if\ x = z\ then\ u\ else\ \sigma\ x)$ **for** x
have $\sigma'\text{-}def'$: $\sigma'\ x = (if\ x \in y\ `\ pp \wedge index\ x = i\ then\ u\ else\ \sigma\ x)$ **for** x
unfolding $\sigma'\text{-}def$ **by** $(rule\ if\text{-}cong,\ insert\ bij\ z,\ auto\ simp: bij\text{-}betw\text{-}def\ inj\text{-}on\text{-}def)$
have $var\text{-}ind\text{-}conv$: $var\text{-}ind\ i\ x = (x = z \vee var\text{-}ind\ (Suc\ i)\ x)$ **for** x
proof
assume $x = z \vee var\text{-}ind\ (Suc\ i)\ x$
thus $var\text{-}ind\ i\ x$ **using** $z\ i$ **unfolding** $var\text{-}ind\text{-}def$ **by** $auto$
next
assume $var\text{-}ind\ i\ x$
hence x: $x \in y\ `\ pp\ index\ x \in \{..<n\} - \{..<i\}$ **unfolding** $var\text{-}ind\text{-}def$ **by** $auto$
with i **have** $index\ x = i \vee index\ x \in \{..<n\} - \{..<Suc\ i\}$ **by** $auto$
thus $x = z \vee var\text{-}ind\ (Suc\ i)\ x$
proof
assume $index\ x = i$
with $x(1)\ z\ bij$ **have** $x = z$ **by** $(auto\ simp: bij\text{-}betw\text{-}def\ inj\text{-}on\text{-}def)$
thus $?thesis$ **by** $auto$
qed $(insert\ x,\ auto\ simp: var\text{-}ind\text{-}def)$
qed
have $[simp]$: $var\text{-}ind\ i\ z$ **unfolding** $var\text{-}ind\text{-}conv$ **by** $auto$
have $[simp]$: $var\text{-}ind\ (Suc\ i)\ z = False$ **unfolding** $var\text{-}ind\text{-}def$ **using** z **by** $auto$
have $\sigma z[simp]$: $\sigma\ z = Var\ z$ **using** $\sigma[unfolded\ cg\text{-}subst\text{-}ind\text{-}def,\ rule\text{-}format,\ of\ z]$ **by** $auto$
have $\sigma'\text{-}upd$: $\sigma' = \sigma(z := u)$ **unfolding** $\sigma'\text{-}def$ **by** $(intro\ ext,\ auto)$

```
have σ'-comp: σ' = σ ∘s Var(z := u) unfolding subst-compose-def σ'-upd
proof (intro ext)
  fix x
  show (σ(z := u)) x = σ x · Var(z := u)
  proof (cases x = z)
    case False
    hence σ x · (Var(z := u)) = σ x · Var
    proof (intro term-subst-eq)
      fix y
      assume y: y ∈ vars (σ x)
      show (Var(z := u)) y = Var y
      proof (cases var-ind i x)
        case True
        with σ[unfolded cg-subst-ind-def, rule-format, of x]
        have σ x = Var x by auto
        with False y show ?thesis by auto
      next
        case False
        with σ[unfolded cg-subst-ind-def, rule-format, of x]
        have vars (σ x) = {} by auto
        with y show ?thesis by auto
      qed
    qed
    thus ?thesis by auto
  qed simp
qed
have σ': cg-subst-ind (Suc i) σ' unfolding cg-subst-ind-def
proof (intro allI conjI impI)
  fix x
  assume var-ind (Suc i) x
  hence var-ind i x and diff: index x ≠ i unfolding var-ind-def by auto
  hence σ x = Var x using σ[unfolded cg-subst-ind-def] by blast
  thus σ' x = Var x unfolding σ'-def' using diff by auto
next
  fix x
  assume ¬ var-ind (Suc i) x and snd x ∈ S
  thus σ' x : snd x in 𝒯(C,EMPTYn)
    using σ[unfolded cg-subst-ind-def, rule-format, of x] u
    unfolding σ'-def var-ind-conv by auto
next
  fix x
  assume ¬ var-ind (Suc i) x
  hence x = z ∨ ¬ var-ind i x unfolding var-ind-conv by auto
  thus vars (σ' x) = {} unfolding σ'-upd using σ[unfolded cg-subst-ind-def,
rule-format, of x] vars-u by auto
qed
show ?case
proof (intro exI[of - σ'] conjI σ' ballI)
  fix mp
```

assume *mp*: $mp \in pp$
define s' **where** $s' = s\ mp \cdot \sigma$
define t' **where** $t' = t\ mp \cdot \sigma$
from *confl*[*rule-format*, *OF mp*]
obtain p **where** p: $p \in poss\ s'$ $p \in poss\ t'$ **and** *confl*: *confl* i $(s'$ |- $p)$ $(t'$ |- $p)$ **by** (*auto simp*: s'-*def* t'-*def*)
{
fix s' t' :: ($'f$, *nat* $\times$ $'s$) *term* **and** p f ss x
assume $*$: $(s'$ |- p, t' |-$p)$ = (*Fun f ss*, *Var x*) *var-ind* i x **and** p: $p \in poss\ s'$ $p \in poss\ t'$
and *range-all-st*: $s' \in$ *all-st*
hence s': $s' \cdot Var(z := u)$ |- p = *Fun f ss* $\cdot$ $Var(z := u)$ (**is** - = *?s*)
and t': $t' \cdot Var(z := u)$ |- p = (*if* $x = z$ *then* u *else* *Var* x) **using** p **by** *auto*
from *range-all-st*[*unfolded all-st-def*]
have *range*σ: $\exists$ S. $s' = S \cdot \sigma$ **by** *auto*
define s **where** s = *?s*
have $\exists p$. $p \in poss\ (s' \cdot Var(z := u)) \wedge p \in poss\ (t' \cdot Var(z := u)) \wedge$ *confl* (*Suc* i) $(s' \cdot Var(z := u)$ |- $p)$ $(t' \cdot Var(z := u)$ |- $p)$
proof (*cases* $x = z$)
case *False*
thus *?thesis* **using** $*$ p **unfolding** s' t' **by** (*intro exI*[*of* - p], *auto simp*: *confl-def var-ind-conv*)
next
case *True*
hence t': $t' \cdot Var(z := u)$ |- $p = u$ **unfolding** t' **by** *auto*
have $\exists$ p'. $p' \in poss\ u \wedge p' \in poss\ s \wedge$ *confl* (*Suc* i) $(s$ |- $p')$ $(u$ |- $p')$
proof (*cases* $\exists$ x. $x \in vars\ s \wedge$ *var-ind* (*Suc* i) x)
case *True*
then obtain x **where** *xs*: $x \in vars\ s$ **and** x: *var-ind* (*Suc* i) x **by** *auto*
from *xs* **obtain** p' **where** p': $p' \in poss\ s$ **and** *sp*: s |- p' = *Var* x **by** (*metis vars-term-poss-subt-at*)
from p' *sp vars-u* **show** *?thesis*
proof (*induct u arbitrary*: p' s)
case (*Fun f us* p' s)
show *?case*
proof (*cases* s)
case (*Var* y)
with *Fun* **have** s: s = *Var* x **by** *auto*
with x **show** *?thesis* **by** (*intro exI*[*of* - *Nil*], *auto simp*: *confl-def*)
next
case s: (*Fun g ss*)
with *Fun* **obtain** j p **where** p: $p' = j$ # p $j <$ *length ss* $p \in poss$ (*ss* ! j) (*ss* ! j) |- p = *Var* x **by** *auto*
show *?thesis*
proof (*cases* (f,*length us*) = (g,*length ss*))
case *False*
thus *?thesis* **by** (*intro exI*[*of* - *Nil*], *auto simp*: s *confl-def*)

next
case *True*
with *p* **have** *j*: $j <$ *length us* **by** *auto*
hence *usj*: *us* ! $j \in$ *set us* **by** *auto*
with *Fun* **have** *vars* (*us* ! *j*) = {} **by** *auto*
from *Fun*(*1*)[*OF usj p*(*3*,*4*) *this*] **obtain** p' **where**
$p' \in$ *poss* (*us* ! *j*) $\wedge$ $p' \in$ *poss* (*ss* ! *j*) $\wedge$ *confl* (*Suc i*) (*ss* ! *j* |- p') (*us* ! *j* |- p') **by** *auto*
thus *?thesis* **using** *j p* **by** (*intro exI*[*of* - *j* # p'], *auto simp*: *s*)
qed
qed
qed *auto*
next
case *False*
from $*$ **have** *fss*: *Fun f ss* = s' |- *p* **by** *auto*
from *range*σ **obtain** *S* **where** *sS*: $s' = S \cdot \sigma$ **by** *auto*
from *p* **have** *vars* (s' |- *p*) $\subseteq$ *vars* s' **by** (*metis vars-term-subt-at*)
also have ... = ($\bigcup y \in$*vars S*. *vars* (σ *y*)) **unfolding** *sS* **by** (*simp add*: *vars-term-subst*)
also have ... $\subseteq$ ($\bigcup y \in$*vars S*. *Collect* (*var-ind i*))
proof −
{
fix *x y*
assume $x \in$ *vars* (σ *y*)
hence *var-ind i x*
using σ[*unfolded cg-subst-ind-def*, *rule-format*, *of y*] **by** *auto*
}
thus *?thesis* **by** *auto*
qed
finally have *sub*: *vars* (s' |- *p*) $\subseteq$ *Collect* (*var-ind i*) **by** *blast*
have *vars s* = *vars* (s' |- $p \cdot$ *Var*($z := u$)) **unfolding** *s-def* s' *fss* **by** *auto*
also have ... = $\bigcup$ (*vars* ‘ *Var*($z := u$) ‘ *vars* (s' |- *p*)) **by** (*simp add*: *vars-term-subst*)
also have ... $\subseteq$ $\bigcup$ (*vars* ‘ *Var*($z := u$) ‘ *Collect* (*var-ind i*)) **using** *sub* **by** *auto*
also have ... $\subseteq$ *Collect* (*var-ind* (*Suc i*))
by (*auto simp*: *vars-u var-ind-conv*)
finally have *vars-s*: *vars s* = {} **using** *False* **by** *auto*

{
assume $s = u$
from *this*[*unfolded s-def fss*]
have *eq*: s' |- $p \cdot$ *Var*($z := u$) = *u* **by** *auto*
have *False*
proof (*cases* $z \in$ *vars* (s' |- *p*))
case *True*
have *diff*: s' |- $p \neq$ *Var z* **using** $*$ **by** *auto*
from *True* **obtain** *C* **where** *id*: s' |- *p* = *C* $\langle$ *Var z* $\rangle$

by (*metis ctxt-supt-id vars-term-poss-subt-at*)
with *diff* **have** *diff*: $C \neq Hole$ **by** (*cases C*, *auto*)
from *eq*[*unfolded id*, *simplified*] *diff*
obtain C **where** $C\langle u\rangle = u$ **and** $C \neq Hole$ **by** (*cases C*; *force*)
from *arg-cong*[*OF this*(*1*), *of size*] *this*(*2*) **show** *False*
by (*simp add*: *less-not-refl2 size-ne-ctxt*)
next
case *False*
have *size*: *size* $s' \in$ *size* ‘ *all-st* **using** *range-all-st* **by** *auto*
from *False* **have** s' |- $p \cdot Var(z := u) = s'$ |- $p \cdot Var$
by (*intro term-subst-eq*, *auto*)
with *eq* **have** *eq*: s' |- $p = u$ **by** *auto*
hence *size* u = *size* (s' |- p) **by** *auto*
also have $\ldots \leq$ *size* s' **using** *p*(*1*)
by (*rule subt-size*)
also have $\ldots \leq$ *Max* (*size* ‘ *all-st*)
using *size fin-all-st* **by** *simp*
also have $\ldots < d$ **unfolding** *d-def* **by** *simp*
also have $\ldots \leq$ *size* u **using** *du* **.**
finally show *False* **by** *simp*
qed
}
hence $s \neq u$ **by** *auto*
with *vars-s vars-u*
show *?thesis*
proof (*induct s arbitrary*: *u*)
case *s*: (*Fun f ss u*)
then obtain *g us* **where** *u*: u = *Fun g us* **by** (*cases u*, *auto*)
show *?case*
proof (*cases* (*f*,*length ss*) = (*g*,*length us*))
case *False*
thus *?thesis* **unfolding** *u* **by** (*intro exI*[*of - Nil*], *auto simp*: *confl-def*)
next
case *True*
with *s*(*4*)[*unfolded u*] **have** $\exists$ $j <$ *length us*. *ss* ! $j \neq$ *us* ! j
by (*auto simp*: *list-eq-nth-eq*)
then obtain j **where** *j*: $j <$ *length us* **and** *diff*: *ss* ! $j \neq$ *us* ! j **by** *auto*
from *j True* **have** *mem*: *ss* ! $j \in$ *set ss us* ! $j \in$ *set us* **by** *auto*
with *s*(*2*−) *u* **have** *vars* (*ss* ! *j*) = {} *vars* (*us* ! *j*) = {} **by** *auto*
from *s*(*1*)[*OF mem*(*1*) *this diff*] **obtain** p' **where**
$p' \in$ *poss* (*us* ! *j*) $\wedge$ $p' \in$ *poss* (*ss* ! *j*) $\wedge$ *confl* (*Suc i*) (*ss* ! *j* |- p') (*us* ! *j* |- p')
by *blast*
thus *?thesis* **unfolding** *u* **using** *True j* **by** (*intro exI*[*of - j # p'*], *auto*)
qed
qed *auto*

qed
then obtain p' **where** p': $p' \in$ *poss u* $p' \in$ *poss s* **and** *confl*: *confl* (*Suc i*) (*s* |- p') (*u* |- p') **by** *auto*
have s'': $s' \cdot$ *Var*(z := u) |- (p @ p') = s |- p' **unfolding** *s-def* s'[*symmetric*] **using** p p' **by** *auto*
have t'': $t' \cdot$ *Var*(z := u) |- (p @ p') = u |- p' **using** t' p p' **by** *auto*
show *?thesis*
proof (*intro exI*[*of* - p @ p'], *unfold* s'' t'', *intro conjI confl*)
have $p \in$ *poss* ($s' \cdot$ *Var*(z := u)) **using** p **by** *auto*
moreover have $p' \in$ *poss* (($s' \cdot$ *Var*(z := u)) |- p) **using** s' p' p **unfolding** *s-def* **by** *auto*
ultimately show p @ $p' \in$ *poss* ($s' \cdot$ *Var*(z := u)) **by** *simp*
have $p \in$ *poss* ($t' \cdot$ *Var*(z := u)) **using** p **by** *auto*
moreover have $p' \in$ *poss* (($t' \cdot$ *Var*(z := u)) |- p) **using** t' p' p **by** *auto*
ultimately show p @ $p' \in$ *poss* ($t' \cdot$ *Var*(z := u)) **by** *simp*
qed
qed
} **note** *main* = *this*
consider (*FF*) *f g ss ts* **where** (s' |- p, t' |- p) = (*Fun f ss*, *Fun g ts*) (f,*length ss*) $\neq$ (g,*length ts*)
| (*FV*) *f ss x* **where** (s' |- p, t' |- p) = (*Fun f ss*, *Var x*) *var-ind i x*
| (*VF*) *f ss x* **where** (s' |- p, t' |- p) = (*Var x*, *Fun f ss*) *var-ind i x*
| (*VV*) x x' **where** (s' |- p, t' |- p) = (*Var x*, *Var* x') $x \neq x'$ *var-ind i x* *var-ind i* x'
using *confl* **by** (*auto simp*: *confl-def split*: *term.splits*)
hence $\exists p.$ $p \in$ *poss* ($s' \cdot$ *Var*(z := u)) $\wedge$ $p \in$ *poss* ($t' \cdot$ *Var*(z := u)) $\wedge$ *confl* (*Suc i*) ($s' \cdot$ *Var*(z := u) |- p) ($t' \cdot$ *Var*(z := u) |- p)
proof *cases*
case (*FF f g ss ts*)
thus *?thesis* **using** p **by** (*intro exI*[*of* - p], *auto simp*: *confl-def*)
next
case (*FV f ss x*)
have $s' \in$ *all-st* **unfolding** s'*-def* **using** *mp all-st-def* **by** *auto*
from *main*[*OF FV p this*] **show** *?thesis* **by** *auto*
next
case (*VF f ss x*)
have t': $t' \in$ *all-st* **unfolding** t'*-def* **using** *mp all-st-def* **by** *auto*
from *VF* **have** (t' |- p, s' |- p) = (*Fun f ss*, *Var x*) *var-ind i x* **by** *auto*
from *main*[*OF this p*(*2*,*1*) t']
obtain p **where** $p \in$ *poss* ($t' \cdot$ *Var*(z := u)) $p \in$ *poss* ($s' \cdot$ *Var*(z := u)) *confl* (*Suc i*) ($t' \cdot$ *Var*(z := u) |- p) ($s' \cdot$ *Var*(z := u) |- p)
by *auto*
thus *?thesis* **by** (*intro exI*[*of* - p], *auto simp*: *confl-def split*: *term.splits*)
next
case (*VV x* x')
thus *?thesis* **using** *p vars-u* **by** (*intro exI*[*of* - p], *cases u*, *auto simp*: *confl-def var-ind-conv*)
qed

thus $\exists p.\ p \in$ *poss* (*s mp* $\cdot\ \sigma'$) $\wedge\ p \in$ *poss* (*t mp* $\cdot\ \sigma'$) $\wedge$ *confl* (*Suc i*) (*s mp* $\cdot\ \sigma'$ |- *p*) (*t mp* $\cdot\ \sigma'$ |- *p*)
unfolding σ'*-comp subst-subst-compose s'-def t'-def* **by** *auto*
qed
qed
}
from *this*[*of n*]
obtain σ **where** σ: *cg-subst-ind n* σ **and** *confl*: $\bigwedge$ *mp. mp* $\in$ *pp* $\Longrightarrow \exists p.\ p \in$ *poss* (*s mp* $\cdot\ \sigma$) $\wedge\ p \in$ *poss* (*t mp* $\cdot\ \sigma$) $\wedge$ *confl n* (*s mp* $\cdot\ \sigma$ |- *p*) (*t mp* $\cdot\ \sigma$ |- *p*)
by *blast*
define σ' :: ($'f$,*nat* $\times$ $'s$,$'v$)*gsubst* **where** σ' *x* = *Var undefined* **for** *x*
let *?*$\sigma = \sigma \circ_s \sigma'$
have *cg-subst ?*σ **unfolding** *cg-subst-def subst-compose-def*
proof (*intro allI impI*)
fix *x* :: *nat* $\times$ $'s$
assume *snd x* $\in$ *S*
with σ[*unfolded cg-subst-ind-def*, *rule-format*, *of x*]
have σ *x* : *snd x in* $\mathcal{T}$(*C*,*EMPTYn*) **by** *auto*
thus σ *x* $\cdot\ \sigma'$: *snd x in* $\mathcal{T}$(*C*,*EMPTY*) **by** (*rule type-conversion1*)
qed
from *pp*[*unfolded pat-complete-def match-complete-wrt-def*, *rule-format*, *OF this*]
obtain *mp* μ **where** *mp*: *mp* $\in$ *pp* **and** *match*: $\bigwedge$ *ti li*. (*ti*, *li*)$\in$ *mp* $\Longrightarrow$ *ti* $\cdot$ *?*σ = *li* $\cdot\ \mu$ **by** *force*
from *P1*[*OF this*(*1*)]
have (*s mp*, *Var* (*x mp*)) $\in$ *mp* (*t mp*, *Var* (*x mp*)) $\in$ *mp* **by** *auto*
from *match*[*OF this*(*1*)] *match*[*OF this*(*2*)] **have** *ident*: *s mp* $\cdot$ *?*σ = *t mp* $\cdot$ *?*σ **by** *auto*
from *confl*[*OF mp*] **obtain** *p*
where *p*: *p* $\in$ *poss* (*s mp* $\cdot\ \sigma$) *p* $\in$ *poss* (*t mp* $\cdot\ \sigma$) **and** *confl*: *confl n* (*s mp* $\cdot\ \sigma$ |- *p*) (*t mp* $\cdot\ \sigma$ |- *p*)
by *auto*
let *?s* = *s mp* $\cdot\ \sigma$ **let** *?t* = *t mp* $\cdot\ \sigma$
from *confl-n*[*OF confl*] **obtain** *f g ss ts* **where**
confl: *?s* |-*p* = *Fun f ss ?t* |-*p* = *Fun g ts* **and** *diff*: (*f*,*length ss*) $\neq$ (*g*,*length ts*) **by** *auto*
define *s'* **where** *s'* = *s mp* $\cdot\ \sigma$
define *t'* **where** *t'* = *t mp* $\cdot\ \sigma$
from *confl p ident*
have *False*
unfolding *subst-subst-compose s'-def*[*symmetric*] *t'-def*[*symmetric*]
proof (*induction p arbitrary*: *s' t'*)
case *Nil*
then show *?case* **using** *diff* **by** (*auto simp*: *list-eq-nth-eq*)
next
case (*Cons i p s t*)
from *Cons* **obtain** *h1 us1* **where** *s*: *s* = *Fun h1 us1* **by** (*cases s*, *auto*)
from *Cons* **obtain** *h2 us2* **where** *t*: *t* = *Fun h2 us2* **by** (*cases t*, *auto*)
from *Cons*(*2*,*4*)[*unfolded s*] **have** *si*: (*us1* ! *i*) |- *p* = *Fun f ss p* $\in$ *poss* (*us1*

! i) **and** *i1*: $i <$ *length us1* **by** *auto*
from *Cons*(*3*,*5*)[*unfolded t*] **have** *ti*: (*us2* ! i) |- p = *Fun g ts p* $\in$ *poss* (*us2* ! i) **and** *i2*: $i <$ *length us2* **by** *auto*
from *Cons*(*6*)[*unfolded s t*] *i1 i2* **have** *us1* ! $i \cdot \sigma'$ = *us2* ! $i \cdot \sigma'$ **by** (*auto simp*: *list-eq-nth-eq*)
from *Cons.IH*[*OF si*(*1*) *ti*(*1*) *si*(*2*) *ti*(*2*) *this*]
show *False* **.**
qed
}
thus *?case* **by** *auto*
qed
end
end

4 A Multiset-Based Inference System to Decide Pattern Completeness

theory *Pattern-Completeness-Multiset*
imports
Pattern-Completeness-Set
LP-Duality.Minimum-Maximum
Polynomial-Factorization.Missing-List
First-Order-Terms.Term-Pair-Multiset
begin

4.1 Definition of the Inference Rules

We next switch to a multiset based implementation of the inference rules. At this level, termination is proven and further, that the evaluation cannot get stuck. The inference rules closely mimic the ones in the paper, though there is one additional inference rule for getting rid of duplicates (which are automatically removed when working on sets).

type-synonym ($'f$,$'v$,$'s$)*match-problem-mset* = (($'f$,*nat* $\times$ $'s$)*term* $\times$ ($'f$,$'v$)*term*) *multiset*
type-synonym ($'f$,$'v$,$'s$)*pat-problem-mset* = ($'f$,$'v$,$'s$)*match-problem-mset multiset*

type-synonym ($'f$,$'v$,$'s$)*pats-problem-mset* = ($'f$,$'v$,$'s$)*pat-problem-mset multiset*

abbreviation *mp-mset* :: ($'f$,$'v$,$'s$)*match-problem-mset* $\Rightarrow$ ($'f$,$'v$,$'s$)*match-problem-set*

where *mp-mset* $\equiv$ *set-mset*

abbreviation *pat-mset* :: ($'f$,$'v$,$'s$)*pat-problem-mset* $\Rightarrow$ ($'f$,$'v$,$'s$)*pat-problem-set*
where *pat-mset* $\equiv$ *image mp-mset o set-mset*

abbreviation *pats-mset* :: ($'f$,$'v$,$'s$)*pats-problem-mset* $\Rightarrow$ ($'f$,$'v$,$'s$)*pats-problem-set*

where *pats-mset* $\equiv$ *image pat-mset o set-mset*

abbreviation (*input*) *bottom-mset* :: $('f,'v,'s)$*pats-problem-mset* **where** *bottom-mset* $\equiv$ {# {#} #}

context *pattern-completeness-context*
begin

A terminating version of $(\Rrightarrow_s)$ working on multisets that also treats the transformation on a more modular basis.

definition *subst-match-problem-mset* :: $('f,nat \times 's)subst \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}mset \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}mset$ **where**
subst-match-problem-mset τ = *image-mset* (*subst-left* τ)

definition *subst-pat-problem-mset* :: $('f,nat \times 's)subst \Rightarrow ('f,'v,'s)pat\text{-}problem\text{-}mset \Rightarrow ('f,'v,'s)pat\text{-}problem\text{-}mset$ **where**
subst-pat-problem-mset τ = *image-mset* (*subst-match-problem-mset* τ)

definition τ*s-list* :: $nat \Rightarrow nat \times 's \Rightarrow ('f,nat \times 's)subst\ list$ **where**
τ*s-list n x* = *map* (τc *n x*) (*Cl* (*snd x*))

inductive *mp-step-mset* :: $('f,'v,'s)match\text{-}problem\text{-}mset \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}mset \Rightarrow bool$ (**infix** $\rightarrow_m$ *50*)**where**
match-decompose: (*f*,*length ts*) = (*g*,*length ls*)
$\Longrightarrow$ *add-mset* (*Fun f ts*, *Fun g ls*) *mp* $\rightarrow_m$ *mp* + *mset* (*zip ts ls*)
| *match-match*: $x \notin \bigcup$ (*vars* ‘ *snd* ‘ *set-mset mp*)
$\Longrightarrow$ *add-mset* (*t*, *Var x*) *mp* $\rightarrow_m$ *mp*
| *match-duplicate*: *add-mset pair* (*add-mset pair mp*) $\rightarrow_m$ *add-mset pair mp*

inductive *match-fail* :: $('f,'v,'s)match\text{-}problem\text{-}mset \Rightarrow bool$ **where**
match-clash: (*f*,*length ts*) $\neq$ (*g*,*length ls*)
$\Longrightarrow$ *match-fail* (*add-mset* (*Fun f ts*, *Fun g ls*) *mp*)
| *match-clash'*: *Conflict-Clash s t* $\Longrightarrow$ *match-fail* (*add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp*))

inductive *pp-step-mset* :: $('f,'v,'s)pat\text{-}problem\text{-}mset \Rightarrow ('f,'v,'s)pats\text{-}problem\text{-}mset \Rightarrow bool$
(**infix** $\Rightarrow_m$ *50*) **where**
pat-remove-pp: *add-mset* {#} *pp* $\Rightarrow_m$ {#}
| *pat-simp-mp*: *mp-step-mset mp mp'* $\Longrightarrow$ *add-mset mp pp* $\Rightarrow_m$ {# (*add-mset mp' pp*) #}
| *pat-remove-mp*: *match-fail mp* $\Longrightarrow$ *add-mset mp pp* $\Rightarrow_m$ {# *pp* #}
| *pat-instantiate*: *tvars-disj-pp* {*n* ..< *n*+*m*} (*pat-mset* (*add-mset mp pp*)) $\Longrightarrow$
(*Var x*, *l*) $\in$ *mp-mset mp* $\wedge$ *is-Fun l* $\vee$
(*s*,*Var y*) $\in$ *mp-mset mp* $\wedge$ (*t*,*Var y*) $\in$ *mp-mset mp* $\wedge$ *Conflict-Var s t x* $\wedge$ $\neg$ *inf-sort* (*snd x*) $\Longrightarrow$
add-mset mp pp $\Rightarrow_m$ *mset* (*map* (λ τ. *subst-pat-problem-mset* τ (*add-mset mp pp*)) (τ*s-list n x*))

inductive *pat-fail* :: $('f,'v,'s)$*pat-problem-mset* $\Rightarrow$ *bool* **where**
  pat-failure': *Ball* (*pat-mset pp*) *inf-var-conflict* $\Longrightarrow$ *pat-fail pp*
| *pat-empty*: *pat-fail* $\{\#\}$

inductive *P-step-mset* :: $('f,'v,'s)$*pats-problem-mset* $\Rightarrow$ $('f,'v,'s)$*pats-problem-mset* $\Rightarrow$ *bool*
  (**infix** $\Rrightarrow_m$ *50*)**where**
  P-failure: *pat-fail pp* $\Longrightarrow$ *add-mset pp P* $\neq$ *bottom-mset* $\Longrightarrow$ *add-mset pp P* $\Rrightarrow_m$ *bottom-mset*
| *P-simp-pp*: *pp* $\Rightarrow_m$ *pp'* $\Longrightarrow$ *add-mset pp P* $\Rrightarrow_m$ *pp'* + *P*

The relation (encoded as predicate) is finally wrapped in a set

definition *P-step* :: $(('f,'v,'s)$*pats-problem-mset* $\times$ $('f,'v,'s)$*pats-problem-mset*$)$*set* $(\Rrightarrow)$ **where**
  $\Rrightarrow$ = $\{(P,P').\ P \Rrightarrow_m P'\}$

4.2 The evaluation cannot get stuck

lemmas *subst-defs* =
  subst-pat-problem-mset-def
  subst-pat-problem-set-def
  subst-match-problem-mset-def
  subst-match-problem-set-def

lemma *pat-mset-fresh-vars*:
  $\exists$ *n*. *tvars-disj-pp* $\{n..<n + m\}$ (*pat-mset p*)
proof –
  define *p'* **where** *p'* = *pat-mset p*
  define *V* **where** *V* = *fst* ‘ $\bigcup$ (*vars* ‘ (*fst* ‘ $\bigcup$ *p'*))
  have *finite V* **unfolding** *V-def p'-def* **by** *auto*
  define *n* **where** *n* = *Suc* (*Max V*)
  {
    fix *mp t l*
    assume *mp* $\in$ *p'* (*t*,*l*) $\in$ *mp*
    hence *sub*: *fst* ‘ *vars t* $\subseteq$ *V* **unfolding** *V-def* **by** *force*
    {
      fix *x*
      assume *x* $\in$ *fst* ‘ *vars t*
      with *sub* **have** *x* $\in$ *V* **by** *auto*
      with ‹*finite V*› **have** *x* $\leq$ *Max V* **by** *simp*
      also have ... < *n* **unfolding** *n-def* **by** *simp*
      finally have *x* < *n* **.**
    }
    hence *fst* ‘ *vars t* $\cap$ $\{n..<n + m\}$ = {} **by** *force*
  }
  thus *?thesis* **unfolding** *tvars-disj-pp-def p'-def*[*symmetric*]
    by (*intro exI*[*of - n*] *ballI*, *force*)
qed

lemma *pat-fail-or-trans*:
pat-fail $p \lor (\exists\ ps.\ p \Rightarrow_m ps)$
proof (*cases* $p = \{\#\}$)
case *True*
with *pat-empty* **show** *?thesis* **by** *auto*
next
case *pne*: *False*
from *pat-mset-fresh-vars* **obtain** n **where** *fresh*: *tvars-disj-pp* $\{n..<n + m\}$ (*pat-mset* p) **by** *blast*
show *?thesis*
proof (*cases* $\{\#\} \in\# p$)
case *True*
then obtain p' **where** $p =$ *add-mset* $\{\#\}$ p' **by** (*rule mset-add*)
with *pat-remove-pp* **show** *?thesis* **by** *auto*
next
case *empty-p*: *False*
show *?thesis*
proof (*cases* $\exists\ mp\ s\ t.\ mp \in\# p \land (s,t) \in\# mp \land$ *is-Fun* t)
case *True*
then obtain *mp s t* **where** *mp*: $mp \in\# p$ **and** $(s,t) \in\# mp$ **and** *is-Fun* t **by** *auto*
then obtain *g ts* **where** *mem*: $(s,$*Fun* $g\ ts) \in\# mp$ **by** (*cases* t, *auto*)
from *mp* **obtain** p' **where** p: $p =$ *add-mset mp* p' **by** (*rule mset-add*)
from *mem* **obtain** mp' **where** *mp*: $mp =$ *add-mset* $(s,$ *Fun* $g\ ts)$ mp' **by** (*rule mset-add*)
show *?thesis*
proof (*cases* s)
case s: (*Fun f ss*)
from *pat-simp-mp*[*OF match-decompose*, *of f ss*] *pat-remove-mp*[*OF match-clash*, *of f ss*]
show *?thesis* **unfolding** *p mp s* **by** *blast*
next
case (*Var x*)
from *Var mem* **obtain** l **where** (*Var* x, l) $\in\# mp \land$ *is-Fun* l **by** *auto*
from *pat-instantiate*[*OF fresh*[*unfolded p*] *disjI1*[*OF this*]]
show *?thesis* **unfolding** p **by** *auto*
qed
next
case *False*
hence *rhs-vars*: $\bigwedge\ mp\ s\ l.\ mp \in\# p \Longrightarrow (s,l) \in\# mp \Longrightarrow$ *is-Var* l **by** *auto*
let *?single-var* $= (\exists\ mp\ t\ x.$ *add-mset* $(t,$*Var* $x)$ $mp \in\# p \land x \notin \bigcup$ (*vars* ‘ *snd* ‘ *set-mset mp*))
let *?duplicate* $= (\exists\ mp\ pair.$ *add-mset pair* (*add-mset pair mp*) $\in\# p)$
show *?thesis*
proof (*cases ?single-var* $\lor$ *?duplicate*)
case *True*
thus *?thesis*
proof

assume *?single-var*
then obtain *mp t x* **where** *mp*: *add-mset* $(t, Var\ x)$ *mp* $\in\#$ *p* **and** *x*: $x \notin \bigcup$ (*vars* ' *snd* ' *set-mset mp*)
by *auto*
from *mp* **obtain** p' **where** p = *add-mset* (*add-mset* $(t, Var\ x)$ *mp*) p' **by** (*rule mset-add*)
with *pat-simp-mp*[*OF match-match*[*OF x*]] **show** *?thesis* **by** *auto*
next
assume *?duplicate*
then obtain *mp pair* **where** *add-mset pair* (*add-mset pair mp*) $\in\#$ *p* (**is** *?dup* $\in\#$ *p*) **by** *auto*
from *mset-add*[*OF this*] **obtain** p' **where**
p: *p* = *add-mset ?dup* p' **.**
from *pat-simp-mp*[*OF match-duplicate*[*of pair*]] **show** *?thesis* **unfolding** *p* **by** *auto*
qed
next
case *False*
hence *ndup*: ¬ *?duplicate* **and** *nsvar*: ¬ *?single-var* **by** *auto*
{
fix *mp*
assume *mpp*: *mp* $\in\#$ *p*
with *empty-p* **have** *mp-e*: *mp* $\neq$ {#} **by** *auto*
obtain *s l* **where** *sl*: (*s*,*l*) $\in\#$ *mp* **using** *mp-e* **by** *auto*
from *rhs-vars*[*OF mpp sl*] *sl* **obtain** *x* **where** *sx*: (*s*, *Var x*) $\in\#$ *mp* **by** (*cases l*, *auto*)
from *mpp* **obtain** p' **where** *p*: *p* = *add-mset mp* p' **by** (*rule mset-add*)
from *sx* **obtain** mp' **where** *mp*: *mp* = *add-mset* (*s*, *Var x*) mp' **by** (*rule mset-add*)
from *nsvar*[*simplified*, *rule-format*, *OF mpp*[*unfolded mp*]]
obtain *t l* **where** (*t*,*l*) $\in\#$ mp' **and** $x \in$ *vars* (*snd* (*t*,*l*)) **by** *force*
with *rhs-vars*[*OF mpp*, *of t l*] **have** *tx*: $(t, Var\ x)$ $\in\#$ mp' **unfolding** *mp* **by** *auto*
then obtain mp'' **where** mp': mp' = *add-mset* (*t*, *Var x*) mp'' **by** (*rule mset-add*)
from *ndup*[*simplified*, *rule-format*] *mpp* **have** $s \neq t$ **unfolding** *mp* mp' **by** *auto*
hence $\exists$ *s t x* mp'. *mp* = *add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) mp') $\wedge$ $s \neq t$ **unfolding** *mp* mp' **by** *auto*
} note *two* = *this*
show *?thesis*
proof (*cases* $\exists$ *mp s t x y*. *add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp*) $\in\#$ *p* $\wedge$ *Conflict-Var s t y* $\wedge$ ¬ *inf-sort* (*snd y*))
case *True*
then obtain *mp s t x y* **where**
mp: *add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp*) $\in\#$ *p* (**is** *?mp* $\in\#$ -) **and** *conf*: *Conflict-Var s t y* **and** *y*: ¬ *inf-sort* (*snd y*)
by *blast*
from *conflicts*(*4*)[*OF conf*] **have** $y \in$ *vars s* $\cup$ *vars t* **by** *auto*

with *mp* **have** $y \in$ *tvars-mp* (*mp-mset ?mp*) **unfolding** *tvars-mp-def* **by** *auto*
from *mp* **obtain** p' **where** *p*: $p =$ *add-mset ?mp* p' **by** (*rule mset-add*)
let *?mp* = *add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp*)
from *pat-instantiate*[*OF - disjI2*, *of n ?mp* p' *s x t y*, *folded p*, *OF fresh*]
show *?thesis* **using** *y conf* **by** *auto*
next
case *no-non-inf*: *False*
show *?thesis*
proof (*cases* ∃ *mp s t x*. *add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp*) ∈# *p* ∧ *Conflict-Clash s t*)
case *True*
then obtain *mp s t x* **where**
mp: *add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp*) ∈# *p* (**is** *?mp* ∈# -) **and** *conf*: *Conflict-Clash s t*
by *blast*
from *pat-remove-mp*[*OF match-clash'*[*OF conf*, *of x mp*]]
show *?thesis* **using** *mset-add*[*OF mp*] **by** *metis*
next
case *no-clash*: *False*
show *?thesis*
proof (*intro disjI1 pat-failure' ballI*)
fix *mp*
assume *mp*: *mp* ∈ *pat-mset p*
then obtain *mp'* **where** *mp'*: *mp'* ∈# *p* **and** *mp*: *mp* = *mp-mset mp'* **by** *auto*
from *two*[*OF mp'*]
obtain *s t x mp''*
where *mp''*: *mp'* = *add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp''*) **and** *diff*: $s \neq t$ **by** *auto*
from *conflicts*(*3*)[*OF diff*] **obtain** *y* **where** *Conflict-Clash s t* ∨ *Conflict-Var s t y* **by** *auto*
with *no-clash mp'' mp'* **have** *conf*: *Conflict-Var s t y* **by** *force*
with *no-non-inf mp'*[*unfolded mp''*] **have** *inf*: *inf-sort* (*snd y*) **by** *blast*
show *inf-var-conflict mp* **unfolding** *inf-var-conflict-def mp mp''*
apply (*rule exI*[*of - s*], *rule exI*[*of - t*])
apply (*intro exI*[*of - x*] *exI*[*of - y*])
using *insert inf conf* **by** *auto*
qed
qed
qed
qed
qed
qed
qed

Pattern problems just have two normal forms: empty set (solvable) or bottom (not solvable)

theorem *P-step-NF*:

assumes *NF*: $P \in NF \Rrightarrow$
shows $P \in \{\{\#\}, \mathit{bottom\text{-}mset}\}$
proof (*rule ccontr*)
assume *nNF*: $P \notin \{\{\#\}, \mathit{bottom\text{-}mset}\}$
from *NF* **have** *NF*: $\neg (\exists\ Q.\ P \Rrightarrow_m Q)$ **unfolding** *P-step-def* **by** *blast*
from *nNF* **obtain** $p\ P'$ **where** *P*: $P = \mathit{add\text{-}mset}\ p\ P'$
using *multiset-cases* **by** *auto*
from *pat-fail-or-trans*
obtain *ps* **where** $\mathit{pat\text{-}fail}\ p \lor p \Rrightarrow_m ps$ **by** *auto*
with *P-simp-pp*[*of p ps*] *NF*
have *pat-fail p* **unfolding** *P* **by** *auto*
from *P-failure*[*OF this*, *of* P', *folded P*] *nNF NF* **show** *False* **by** *auto*
qed
end

4.3 Termination

A measure to count the number of function symbols of the first argument that don't occur in the second argument

fun *fun-diff* :: $('f,'v)\mathit{term} \Rightarrow ('f,'w)\mathit{term} \Rightarrow \mathit{nat}$ **where**
fun-diff l (*Var x*) = *num-funs l*
| *fun-diff* (*Fun g ls*) (*Fun f ts*) = (*if f* = *g* $\land$ *length ts* = *length ls then*
sum-list (*map2 fun-diff ls ts*) *else 0*)
| *fun-diff l t* = *0*

lemma *fun-diff-Var*[*simp*]: *fun-diff* (*Var x*) *t* = *0*
by (*cases t*, *auto*)

lemma *add-many-mult*: $(\bigwedge\ y.\ y \in\# N \Longrightarrow (y,x) \in R) \Longrightarrow (N + M,\ \mathit{add\text{-}mset}\ x\ M) \in \mathit{mult}\ R$
by (*metis add.commute add-mset-add-single multi-member-last multi-self-add-other-not-self one-step-implies-mult*)

lemma *fun-diff-num-funs*: *fun-diff l t* $\leq$ *num-funs l*
proof (*induct l t rule*: *fun-diff.induct*)
case (*2 f ls g ts*)
show *?case*
proof (*cases f* = *g* $\land$ *length ts* = *length ls*)
case *True*
have *sum-list* (*map2 fun-diff ls ts*) $\leq$ *sum-list* (*map num-funs ls*)
by (*intro sum-list-mono2*, *insert True 2*, (*force simp*: *set-zip*)+)
with *2* **show** *?thesis* **by** *auto*
qed *auto*
qed *auto*

lemma *fun-diff-subst*: *fun-diff l* $(t \cdot \sigma) \leq$ *fun-diff l t*
proof (*induct l arbitrary*: *t*)
case *l*: (*Fun f ls*)
show *?case*

proof (*cases t*)
  case *t*: (*Fun g ts*)
  show *?thesis* **unfolding** *t* **using** *l* **by** (*auto intro*: *sum-list-mono2*)
next
  case *t*: (*Var x*)
  show *?thesis* **unfolding** *t* **using** *fun-diff-num-funs*[*of Fun f ls*] **by** *auto*
qed
qed *auto*

lemma *fun-diff-num-funs-lt*: **assumes** t': $t' = $ *Fun c cs*
  and *is-Fun l*
shows *fun-diff l* t' < *num-funs l*
proof −
  from *assms* **obtain** *g ls* **where** *l*: *l* = *Fun g ls* **by** (*cases l*, *auto*)
  show *?thesis*
  proof (*cases c* = *g* ∧ *length cs* = *length ls*)
    case *False*
    thus *?thesis* **unfolding** t' *l* **by** *auto*
  next
    case *True*
    have *sum-list* (*map2 fun-diff ls cs*) ≤ *sum-list* (*map num-funs ls*)
      apply (*rule sum-list-mono2*; (*intro impI*)*?*)
      subgoal using *True* **by** *auto*
      subgoal for *i* **using** *True* **by** (*auto intro*: *fun-diff-num-funs*)
      done
    thus *?thesis* **unfolding** t' *l* **using** *True* **by** *auto*
  qed
qed

lemma *sum-union-le-nat*: *sum* (*f* :: *'a* ⇒ *nat*) (*A* ∪ *B*) ≤ *sum f A* + *sum f B*
  by (*metis finite-Un le-iff-add sum.infinite sum.union-inter zero-le*)

lemma *sum-le-sum-list-nat*: *sum f* (*set xs*) ≤ (*sum-list* (*map f xs*) :: *nat*)
proof (*induct xs*)
  case (*Cons x xs*)
  thus *?case*
    by (*cases x* ∈ *set xs*, *auto simp*: *insert-absorb*)
qed *auto*

lemma *bdd-above-has-Maximum-nat*: *bdd-above* (*A* :: *nat set*) ⟹ *A* ≠ {} ⟹ *has-Maximum A*
  unfolding *has-Maximum-def*
  by (*meson Max-ge Max-in bdd-above-nat*)

context *pattern-completeness-context-with-assms*
begin

lemma *τs-list*: *set* (*τs-list n x*) = *τs n x*

unfolding *τs-list-def τs-def* **using** *Cl* **by** *auto*

abbreviation (*input*) *sum-ms* :: ($'a \Rightarrow nat$) $\Rightarrow$ $'a$ *multiset* $\Rightarrow$ *nat* **where**
sum-ms f ms $\equiv$ *sum-mset* (*image-mset f ms*)

definition *meas-diff* :: ($'f$,$'v$,$'s$)*pat-problem-mset* $\Rightarrow$ *nat* **where**
meas-diff = *sum-ms* (*sum-ms* (λ (*t*,*l*). *fun-diff l t*))

definition *max-size* :: $'s \Rightarrow nat$ **where**
max-size s = (*if s* $\in$ *S* $\wedge$ $\neg$ *inf-sort s then Maximum* (*size* ' {*t. t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*)}) *else 0*)

definition *meas-finvars* :: ($'f$,$'v$,$'s$)*pat-problem-mset* $\Rightarrow$ *nat* **where**
meas-finvars = *sum-ms* (λ *mp. sum* (*max-size o snd*) (*tvars-mp* (*mp-mset mp*)))

definition *meas-symbols* :: ($'f$,$'v$,$'s$)*pat-problem-mset* $\Rightarrow$ *nat* **where**
meas-symbols = *sum-ms size-mset*

definition *meas-setsize* :: ($'f$,$'v$,$'s$)*pat-problem-mset* $\Rightarrow$ *nat* **where**
meas-setsize p = *sum-ms* (*sum-ms* (λ -. *1*)) *p* + *size p*

definition *rel-pat* :: (($'f$,$'v$,$'s$)*pat-problem-mset* $\times$ ($'f$,$'v$,$'s$)*pat-problem-mset*)*set* ($\prec$)
where
($\prec$) = *inv-image* ({(*x*, *y*). *x* < *y*} <∗*lex*∗> {(*x*, *y*). *x* < *y*} <∗*lex*∗> {(*x*, *y*). *x* < *y*} <∗*lex*∗> {(*x*, *y*). *x* < *y*})
(λ *mp.* (*meas-diff mp*, *meas-finvars mp*, *meas-symbols mp*, *meas-setsize mp*))

abbreviation *gt-rel-pat* (**infix** $\succ$ *50*) **where**
pp $\succ$ *pp'* $\equiv$ (*pp'*,*pp*) $\in$ $\prec$

definition *rel-pats* :: (($'f$,$'v$,$'s$)*pats-problem-mset* $\times$ ($'f$,$'v$,$'s$)*pats-problem-mset*)*set* ($\prec$*mul*) **where**
$\prec$*mul* = *mult* ($\prec$)

abbreviation *gt-rel-pats* (**infix** $\succ$*mul 50*) **where**
P $\succ$*mul P'* $\equiv$ (*P'*,*P*) $\in$ $\prec$*mul*

lemma *wf-rel-pat*: *wf* $\prec$
unfolding *rel-pat-def*
by (*intro wf-inv-image wf-lex-prod wf-less*)

lemma *wf-rel-pats*: *wf* $\prec$*mul*
unfolding *rel-pats-def*
by (*intro wf-inv-image wf-mult wf-rel-pat*)

lemma *tvars-mp-fin*:
finite (*tvars-mp* (*mp-mset mp*))
unfolding *tvars-mp-def* **by** *auto*

lemmas *meas-def* = *meas-finvars-def meas-diff-def meas-symbols-def meas-setsize-def*

lemma *tvars-mp-mono*: $mp \subseteq\# \ mp' \Longrightarrow$ *tvars-mp* (*mp-mset mp*) $\subseteq$ *tvars-mp* (*mp-mset mp'*)
  unfolding *tvars-mp-def*
  by (*intro image-mono subset-refl set-mset-mono UN-mono*)

lemma *meas-finvars-mono*: **assumes** *tvars-mp* (*mp-mset mp*) $\subseteq$ *tvars-mp* (*mp-mset mp'*)
  shows *meas-finvars* {#*mp*#} $\leq$ *meas-finvars* {#*mp'*#}
  using *tvars-mp-fin*[*of mp'*] *assms*
  unfolding *meas-def* **by** (*auto intro*: *sum-mono2*)

lemma *rel-mp-sub*: {# *add-mset p mp*#} $\succ$ {# *mp* #}
proof −
  let *?mp'* = *add-mset p mp*
  have *mp* $\subseteq\#$ *?mp'* **by** *auto*
  from *meas-finvars-mono*[*OF tvars-mp-mono*[*OF this*]]
  show *?thesis* **unfolding** *meas-def rel-pat-def* **by** *auto*
qed

lemma *rel-mp-mp-step-mset*:
  assumes $mp \rightarrow_m mp'$
  shows {#*mp*#} $\succ$ {#*mp'*#}
  using *assms*
proof *cases*
  case ∗: (*match-decompose f ts g ls mp''*)
  have *meas-finvars* {#*mp'*#} $\leq$ *meas-finvars* {#*mp*#}
  proof (*rule meas-finvars-mono*)
    show *tvars-mp* (*mp-mset mp'*) $\subseteq$ *tvars-mp* (*mp-mset mp*)
      unfolding *tvars-mp-def* ∗ **using** ∗(*3*) **by** (*auto simp*: *set-zip set-conv-nth*)
  qed
  moreover
  have *id*: (*case case x of* (*x*, *y*) ⇒ (*y*, *x*) *of* (*t*, *l*) ⇒ *f t l*) = (*case x of* (*a*,*b*) ⇒ *f b a*) **for**
    x :: (*'f*, *'v*) *Term.term* × (*'f*, *nat* × *'s*) *Term.term* **and** *f* :: - ⇒ - ⇒ *nat*
    by (*cases x*, *auto*)
  have *meas-diff* {#*mp'*#} $\leq$ *meas-diff* {#*mp*#}
    unfolding *meas-def* ∗ **using** ∗(*3*)
   by (*auto simp*: *sum-mset-sum-list*[*symmetric*] *zip-commute*[*of ts ls*] *image-mset.compositionality o-def id*)
  moreover have *meas-symbols* {#*mp'*#} < *meas-symbols* {#*mp*#}
    unfolding *meas-def* ∗ **using** ∗(*3*) *size-mset-Fun-less*[*of ts ls g g*]
    by (*auto simp*: *sum-mset-sum-list*)
  ultimately show *?thesis* **unfolding** *rel-pat-def* **by** *auto*
next
  case ∗: (*match-match x t*)
  show *?thesis* **unfolding** ∗

by (*rule rel-mp-sub*)
next
case $*$: (*match-duplicate pair mp*)
show *?thesis* **unfolding** $*$
by (*rule rel-mp-sub*)
qed

lemma *sum-ms-image*: *sum-ms f* (*image-mset g ms*) = *sum-ms* (*f o g*) *ms*
by (*simp add*: *multiset.map-comp*)

lemma *meas-diff-subst-le*: *meas-diff* (*subst-pat-problem-mset* τ *p*) $\leq$ *meas-diff p*
unfolding *meas-def subst-match-problem-set-def subst-defs subst-left-def*
unfolding *sum-ms-image o-def*
apply (*rule sum-mset-mono*, *rule sum-mset-mono*)
apply *clarify*
unfolding *map-prod-def split id-apply*
by (*rule fun-diff-subst*)

lemma *meas-sub*: **assumes** *sub*: $p' \subseteq\# \ p$
shows *meas-diff* $p' \leq$ *meas-diff p*
meas-finvars $p' \leq$ *meas-finvars p*
meas-symbols $p' \leq$ *meas-symbols p*
proof $-$
from *sub* **obtain** p'' **where** *p*: $p = p' + p''$ **by** (*metis subset-mset.less-eqE*)
show *meas-diff* $p' \leq$ *meas-diff p meas-finvars* $p' \leq$ *meas-finvars p meas-symbols* $p' \leq$ *meas-symbols p*
unfolding *meas-def p* **by** *auto*
qed

lemma *meas-sub-rel-pat*: **assumes** *sub*: $p' \subset\# \ p$
shows $p \succ p'$
proof $-$
from *sub* **obtain** $x \ p''$ **where** *p*: p = *add-mset* $x \ p' + p''$
by (*metis multi-nonempty-split subset-mset.lessE union-mset-add-mset-left union-mset-add-mset-right*)
hence *lt*: *meas-setsize* p' < *meas-setsize p* **unfolding** *meas-def* **by** *auto*
from *sub* **have** $p' \subseteq\# \ p$ **by** *auto*
from *lt meas-sub*[*OF this*]
show *?thesis* **unfolding** *rel-pat-def* **by** *auto*
qed

lemma *max-size-term-of-sort*: **assumes** *sS*: $s \in S$ **and** *inf*: $\neg$ *inf-sort s*
shows $\exists$ *t*. *t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*) $\wedge$ *max-size s* = *size t* $\wedge$ ($\forall$ t'. t' : *s in* $\mathcal{T}$(*C*,*EMPTYn*) $\longrightarrow$ *size* $t' \leq$ *size t*)
proof $-$
let *?set* = λ *s*. *size* ' {*t*. *t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*)}
have *m*: *max-size s* = *Maximum* (*?set s*) **unfolding** *o-def max-size-def* **using** *inf sS* **by** *auto*

from *inf*[*unfolded inf-sort-def*[*OF sS*]] **have** *bdd-above* (*?set s*) **by** *auto*

moreover from *sorts-non-empty*[*OF sS*] *type-conversion2* **have** *?set s* $\neq$ {} **by** *auto*
ultimately have *has-Maximum* (*?set s*) **by** (*rule bdd-above-has-Maximum-nat*)
from *has-MaximumD*[*OF this*, *folded m*] **show** *?thesis* **by** *auto*
qed

lemma *max-size-max*: **assumes** *sS*: *s* $\in$ *S*
and *inf*: $\neg$ *inf-sort s*
and *sort*: *t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*)
shows *size t* $\leq$ *max-size s*
using *max-size-term-of-sort*[*OF sS inf*] *sort* **by** *auto*

lemma *finite-sort-size*: **assumes** *c*: *c* : *map snd vs* $\rightarrow$ *s in C*
and *inf*: $\neg$ *inf-sort s*
shows *sum* (*max-size o snd*) (*set vs*) < *max-size s*
proof –
from *c* **have** *vsS*: *insert s* (*set* (*map snd vs*)) $\subseteq$ *S* **using** *C-sub-S*
by (*metis* (*mono-tags*))
hence *sS*: *s* $\in$ *S* **by** *auto*
let *?m* = *max-size s*
show *?thesis*
proof (*cases* $\exists$ *v* $\in$ *set vs*. *inf-sort* (*snd v*))
case *True*
{
fix *v*
assume *v* $\in$ *set vs*
with *vsS* **have** *v*: *snd v* $\in$ *S* **by** *auto*
note *term-of-sort*[*OF this*]
}
hence $\forall$ *v*. $\exists$ *t*. *v* $\in$ *set vs* $\longrightarrow$ *t* : *snd v in* $\mathcal{T}$(*C*,*EMPTYn*) **by** *auto*
from *choice*[*OF this*] **obtain** *t* **where**
t: $\bigwedge$ *v*. *v* $\in$ *set vs* $\Longrightarrow$ *t v* : *snd v in* $\mathcal{T}$(*C*,*EMPTYn*) **by** *blast*
from *True vsS* **obtain** *vl* **where** *vl*: *vl* $\in$ *set vs* **and** *vlS*: *snd vl* $\in$ *S* **and** *inf-vl*: *inf-sort* (*snd vl*) **by** *auto*
from *not-bdd-above-natD*[*OF inf-vl*[*unfolded inf-sort-def*[*OF vlS*]], *of ?m*] *t*[*OF vl*]
obtain *tl* **where**
tl: *tl* : *snd vl in* $\mathcal{T}$(*C*,*EMPTYn*) **and** *large*: *?m* $\leq$ *size tl* **by** *fastforce*
let *?t* = *Fun c* (*map* (λ *v*. *if v* = *vl then tl else t v*) *vs*)
have *?t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*)
by (*intro Fun-hastypeI*[*OF c*] *list-all2-map-map*, *insert tl t*, *auto*)
from *max-size-max*[*OF sS inf this*]
have *False* **using** *large split-list*[*OF vl*] **by** *auto*
thus *?thesis* **..**
next
case *False*
{
fix *v*
assume *v*: *v* $\in$ *set vs*

with *False* **have** *inf*: ¬ *inf-sort* (*snd v*) **by** *auto*
from *vsS v* **have** *snd v* ∈ *S* **by** *auto*
from *max-size-term-of-sort*[*OF this inf*]
have ∃ *t*. *t* : *snd v in* $\mathcal{T}$(*C*,*EMPTYn*) ∧ *size t* = *max-size* (*snd v*) **by** *auto*
}
hence ∀ *v*. ∃ *t*. *v* ∈ *set vs* ⟶ *t* : *snd v in* $\mathcal{T}$(*C*,*EMPTYn*) ∧ *size t* = *max-size* (*snd v*) **by** *auto*
from *choice*[*OF this*] **obtain** *t* **where**
t: *v* ∈ *set vs* ⟹ *t v* : *snd v in* $\mathcal{T}$(*C*,*EMPTYn*) ∧ *size* (*t v*) = *max-size* (*snd v*) **for** *v* **by** *blast*
let *?t* = *Fun c* (*map t vs*)
have *?t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*)
by (*intro Fun-hastypeI*[*OF c*] *list-all2-map-map*, *insert t*, *auto*)
from *max-size-max*[*OF sS inf this*]
have *size ?t* ≤ *max-size s* **.**

have *sum* (*max-size* ∘ *snd*) (*set vs*) = *sum* (*size o t*) (*set vs*)
by (*rule sum.cong*[*OF refl*], *unfold o-def*, *insert t*, *auto*)
also have . . . ≤ *sum-list* (*map* (*size o t*) *vs*)
by (*rule sum-le-sum-list-nat*)
also have . . . ≤ *size-list* (*size o t*) *vs* **by** (*induct vs*, *auto*)
also have . . . < *size ?t* **by** *simp*
also have . . . ≤ *max-size s* **by** *fact*
finally show *?thesis* **.**
qed
qed

lemma *rel-pp-step-mset*:
assumes $p \Rightarrow_m ps$
and *p′* ∈# *ps*
shows *p* ≻ *p′*
using *assms*
proof *induct*
case ∗: (*pat-simp-mp mp mp′ p*)
hence *p′*: *p′* = *add-mset mp′ p* **by** *auto*
from *rel-mp-mp-step-mset*[*OF* ∗(*1*)]
show *?case* **unfolding** *p′ rel-pat-def meas-def* **by** *auto*
next
case (*pat-remove-mp mp p*)
hence *p′*: *p′* = *p* **by** *auto*
show *?case* **unfolding** *p′*
by (*rule meas-sub-rel-pat*, *auto*)
next
case ∗: (*pat-instantiate n mp p x l s y t*)
from ∗(*2*) **have** ∃ *s t*. (*s*,*t*) ∈# *mp* ∧ (*s* = *Var x* ∧ *is-Fun t*
∨ (*x* ∈ *vars s* ∧ ¬ *inf-sort* (*snd x*)))
proof
assume ∗: (*s*, *Var y*) ∈# *mp* ∧ (*t*, *Var y*) ∈# *mp* ∧ *Conflict-Var s t x* ∧ ¬ *inf-sort* (*snd x*)

hence *Conflict-Var s t x* **and** ¬ *inf-sort* (*snd x*) **by** *auto*
from *conflicts*(*4*)[*OF this*(*1*)] *this*(*2*) ∗
show *?thesis* **by** *auto*
qed *auto*
then obtain *s t* **where** *st*: (*s*,*t*) ∈# *mp* **and** *choice*: *s* = *Var x* ∧ *is-Fun t* ∨ *x* ∈ *vars s* ∧ ¬ *inf-sort* (*snd x*)
by *auto*
let *?p* = *add-mset mp p*
let *?s* = *snd x*
from ∗(*3*) *τs-list*
obtain *τ* **where** *τ*: *τ* ∈ *τs n x* **and** *p′*: *p′* = *subst-pat-problem-mset τ ?p* **by** *auto*

let *?tau-mset* = *subst-pat-problem-mset τ*
let *?tau* = *subst-match-problem-mset τ*
from *τ*[*unfolded τs-def τc-def List.maps-def*]
obtain *c sorts* **where** *c*: *c* : *sorts* → *?s in C* **and** *tau*: *τ* = *subst x* (*Fun c* (*map Var* (*zip* [*n*..<*n* + *length sorts*] *sorts*)))
by *auto*
with *C-sub-S* **have** *sS*: *?s* ∈ *S* **and** *sorts*: *set sorts* ⊆ *S* **by** *auto*
define *vs* **where** *vs* = *zip* [*n*..<*n* + *length sorts*] *sorts*
have *τ*: *τ* = *subst x* (*Fun c* (*map Var vs*)) **unfolding** *tau vs-def* **by** *auto*
have *snd* ‘ *vars* (*τ y*) ⊆ *insert* (*snd y*) *S* **for** *y*
using *sorts* **unfolding** *tau* **by** (*auto simp*: *subst-def set-zip set-conv-nth*)
hence *vars-sort*: (*a*,*b*) ∈ *vars* (*τ y*) ⟹ *b* ∈ *insert* (*snd y*) *S* **for** *a b y* **by** *fastforce*

from *st* **obtain** *mp′* **where** *mp*: *mp* = *add-mset* (*s*,*t*) *mp′* **by** (*rule mset-add*)
from *choice* **have** *?p* ≻ *?tau-mset ?p*
proof
assume *s* = *Var x* ∧ *is-Fun t*
then obtain *f ts* **where** *s*: *s* = *Var x* **and** *t*: *t* = *Fun f ts* **by** (*cases t*, *auto*)
have *meas-diff* (*?tau-mset ?p*) =
meas-diff (*?tau-mset* (*add-mset mp′ p*)) + *fun-diff t* (*s* · *τ*)
unfolding *meas-def subst-defs subst-left-def mp* **by** *simp*
also have . . . ≤ *meas-diff* (*add-mset mp′ p*) + *fun-diff t* (*τ x*) **using** *meas-diff-subst-le*[*of τ*] *s* **by** *auto*
also have . . . < *meas-diff* (*add-mset mp′ p*) + *fun-diff t s*
proof (*rule add-strict-left-mono*)
have *fun-diff t* (*τ x*) < *num-funs t*
unfolding *tau subst-simps fun-diff.simps*
by (*rule fun-diff-num-funs-lt*[*OF refl*], *auto simp*: *t*)
thus *fun-diff t* (*τ x*) < *fun-diff t s* **by** (*auto simp*: *s t*)
qed
also have . . . = *meas-diff ?p* **unfolding** *mp meas-def* **by** *auto*
finally show *?thesis* **unfolding** *rel-pat-def* **by** *auto*
next
assume *x* ∈ *vars s* ∧ ¬ *inf-sort* (*snd x*)
hence *x*: *x* ∈ *vars s* **and** *inf*: ¬ *inf-sort* (*snd x*) **by** *auto*
from *meas-diff-subst-le*[*of τ*]
have *fd*: *meas-diff p′* ≤ *meas-diff ?p* **unfolding** *p′* **.**

have *meas-finvars* (*?tau-mset ?p*) = *meas-finvars* (*?tau-mset* {#*mp*#}) + *meas-finvars* (*?tau-mset p*)
unfolding *subst-defs meas-def* **by** *auto*
also have ... < *meas-finvars* {#*mp*#} + *meas-finvars p*
proof (*rule add-less-le-mono*)
have *vars-τ-var*: *vars* (τ *y*) = (*if x* = *y then set vs else* {*y*}) **for** *y* **unfolding** τ *subst-def* **by** *auto*
have *vars-τ*: *vars* (*t* · τ) = *vars t* − {*x*} ∪ (*if x* ∈ *vars t then set vs else* {}) **for** *t*
unfolding *vars-term-subst image-comp o-def vars-τ-var* **by** *auto*
have *tvars-mp-subst*: *tvars-mp* (*mp-mset* (*?tau mp*)) =
tvars-mp (*mp-mset mp*) − {*x*} ∪ (*if x* ∈ *tvars-mp* (*mp-mset mp*) *then set vs else* {}) **for** *mp*
unfolding *subst-defs subst-left-def tvars-mp-def*
by (*auto simp*:*vars-τ split*: *if-splits prod.split*)
have *id1*: *meas-finvars* (*?tau-mset* {#*mp*#}) = ($\sum$ *x*∈ *tvars-mp* (*mp-mset* (*?tau mp*)). *max-size* (*snd x*)) **for** *mp*
unfolding *meas-def subst-defs* **by** *auto*
have *id2*: *meas-finvars* {#*mp*#} = ($\sum$ *x*∈*tvars-mp* (*mp-mset mp*). *max-size* (*snd x*)) **for** *mp*
unfolding *meas-def subst-defs* **by** *simp*
have *eq*: *x* ∉ *tvars-mp* (*mp-mset mp*) ⟹ *meas-finvars* (*?tau-mset* {# *mp* #}) = *meas-finvars* {#*mp*#} **for** *mp*
unfolding *id1 id2* **by** (*rule sum.cong*[*OF* - *refl*], *auto simp*: *tvars-mp-subst*)
{
fix *mp*

assume *xmp*: *x* ∈ *tvars-mp* (*mp-mset mp*)
let *?mp* = (*mp-mset mp*)
have *fin*: *finite* (*tvars-mp ?mp*) **by** (*rule tvars-mp-fin*)
define *Mp* **where** *Mp* = *tvars-mp ?mp* − {*x*}
from *xmp* **have** *1*: *tvars-mp* (*mp-mset* (*?tau mp*)) = *set vs* ∪ *Mp*
unfolding *tvars-mp-subst Mp-def* **by** *auto*
from *xmp* **have** *2*: *tvars-mp ?mp* = *insert x Mp* **and** *xMp*: *x* ∉ *Mp* **unfolding** *Mp-def* **by** *auto*
from *fin* **have** *fin*: *finite Mp* **unfolding** *Mp-def* **by** *auto*
have *meas-finvars* (*?tau-mset* {# *mp* #}) = *sum* (*max-size* ∘ *snd*) (*set vs* ∪ *Mp*) (**is** - = *sum ?size* -)
unfolding *id1 id2* **using** *1* **by** *auto*
also have ... ≤ *sum ?size* (*set vs*) + *sum ?size Mp* **by** (*rule sum-union-le-nat*)
also have ... < *?size x* + *sum ?size Mp*
proof −
have *sS*: *?s* ∈ *S* **by** *fact*
have *sorts*: *sorts* = *map snd vs* **unfolding** *vs-def* **by** (*intro nth-equalityI*, *auto*)
have *sum ?size* (*set vs*) < *?size x*
using *finite-sort-size*[*OF c*[*unfolded sorts*] *inf*] **by** *auto*
thus *?thesis* **by** *auto*
qed

also have ... = *meas-finvars* $\{\#mp\#\}$ **unfolding** *id2 2* **using** *fin xMp* **by** *auto*
finally have *meas-finvars* (*?tau-mset* $\{\#\ mp\ \#\}$) $<$ *meas-finvars* $\{\#mp\#\}$ **.**
} **note** *less* = *this*
have *le*: *meas-finvars* (*?tau-mset* $\{\#\ mp\ \#\}$) $\leq$ *meas-finvars* $\{\#mp\#\}$ **for** *mp*
using *eq*[*of mp*] *less*[*of mp*] **by** *linarith*

show *meas-finvars* (*?tau-mset* $\{\#mp\#\}$) $<$ *meas-finvars* $\{\#mp\#\}$ **using** *x*
by (*intro less*, *unfold mp*, *force simp*: *tvars-mp-def*)

show *meas-finvars* (*?tau-mset p*) $\leq$ *meas-finvars p*
unfolding *subst-pat-problem-mset-def meas-finvars-def sum-ms-image o-def*
apply (*rule sum-mset-mono*)
subgoal for *mp* **using** *le*[*of mp*] **unfolding** *meas-finvars-def o-def subst-defs* **by** *auto*
done
qed
also have ... = *meas-finvars ?p* **unfolding** p' *meas-def* **by** *simp*
finally show *?thesis* **using** *fd* **unfolding** *rel-pat-def* p' **by** *auto*
qed
thus *?case* **unfolding** p' **.**
next
case $*$: (*pat-remove-pp p*)
thus *?case* **by** (*intro meas-sub-rel-pat*, *auto*)
qed

finally: the transformation is terminating w.r.t. ($\succ$*mul*)

lemma *rel-P-trans*:
assumes $P \Rrightarrow_m P'$
shows $P \succ mul\ P'$
using *assms*
proof *induct*
case $*$: (*P-failure p P*)
from $*$ **have** $p \neq \{\#\} \vee p = \{\#\} \wedge P \neq \{\#\}$ **by** *auto*
thus *?case*
proof
assume $p \neq \{\#\}$
then obtain *mp* p' **where** *p*: p = *add-mset mp* p' **by** (*cases p*, *auto*)
have $p \succ \{\#\}$ **unfolding** *p* **by** (*intro meas-sub-rel-pat*, *auto*)
thus *?thesis* **unfolding** *rel-pats-def* **using**
one-step-implies-mult[*of add-mset p P* $\{\#\{\#\}\#\}$ - $\{\#\}$]
by *auto*
next
assume $*$: $p = \{\#\} \wedge P \neq \{\#\}$ **then obtain** p' P' **where** *p*: $p = \{\#\}$ **and** *P*: P = *add-mset* p' P' **by** (*cases P*, *auto*)
show *?thesis* **unfolding** *P p* **unfolding** *rel-pats-def*
by (*simp add*: *subset-implies-mult*)

qed
next
case $*$: (*P-simp-pp p ps P*)
from *rel-pp-step-mset*[*OF* $*$]
show *?case* **unfolding** *rel-pats-def* **by** (*metis add-many-mult*)
qed

termination of the multiset based implementation

theorem *SN-P-step*: $SN \Rrightarrow$
proof $-$
have *sub*: $\Rrightarrow \subseteq \prec mul\widehat{\ }{-1}$
using *rel-P-trans* **unfolding** *P-step-def* **by** *auto*
show *?thesis*
apply (*rule SN-subset*[*OF - sub*])
using *wf-rel-pats* **by** (*simp add*: *wf-imp-SN*)
qed

4.4 Partial Correctness via Refinement

Obtain partial correctness via a simulation property, that the multiset-based implementation is a refinement of the set-based implementation.

lemma *mp-step-cong*: $mp1 \rightarrow_s mp2 \Longrightarrow mp1 = mp1' \Longrightarrow mp2 = mp2' \Longrightarrow mp1' \rightarrow_s mp2'$ **by** *auto*

lemma *mp-step-mset-mp-trans*: $mp \rightarrow_m mp' \Longrightarrow$ *mp-mset mp* $\rightarrow_s$ *mp-mset mp'*
proof (*induct mp mp' rule*: *mp-step-mset.induct*)
case $*$: (*match-decompose f ts g ls mp*)
show *?case* **by** (*rule mp-step-cong*[*OF mp-decompose*], *insert* $*$, *auto*)
next
case $*$: (*match-match x mp t*)
show *?case* **by** (*rule mp-step-cong*[*OF mp-match*], *insert* $*$, *auto*)
next
case (*match-duplicate pair mp*)
show *?case* **by** (*rule mp-step-cong*[*OF mp-identity*], *auto*)
qed

lemma *mp-fail-cong*: *mp-fail mp* $\Longrightarrow mp = mp' \Longrightarrow$ *mp-fail mp'* **by** *auto*

lemma *match-fail-mp-fail*: *match-fail mp* $\Longrightarrow$ *mp-fail* (*mp-mset mp*)
proof (*induct mp rule*: *match-fail.induct*)
case $*$: (*match-clash f ts g ls mp*)
show *?case* **by** (*rule mp-fail-cong*[*OF mp-clash*], *insert* $*$, *auto*)
next
case $*$: (*match-clash' s t x mp*)
show *?case* **by** (*rule mp-fail-cong*[*OF mp-clash'*], *insert* $*$, *auto*)
qed

lemma *P-step-set-cong*: $P \Rrightarrow_s Q \Longrightarrow P = P' \Longrightarrow Q = Q' \Longrightarrow P' \Rrightarrow_s Q'$ **by** *auto*

lemma *P-step-mset-imp-set*: **assumes** $P \Rrightarrow_m Q$
  shows *pats-mset* $P \Rrightarrow_s$ *pats-mset* Q
  using *assms*
proof (*induct*)
  case $*$: (*P-failure p P*)
  let *?P* = *insert* (*pat-mset p*) (*pats-mset P*)
  from $*$(*1*)
  have *?P* $\Rrightarrow_s$ *bottom*
  proof *induct*
    case (*pat-failure′ p*)
    from *P-failure′*[*OF this*]
    show *?case* **by** *auto*
  next
    case *pat-empty*
    show *?case* **using** *P-fail* **by** *auto*
  qed
  thus *?case* **by** *auto*
next
  case $*$: (*P-simp-pp p ps P*)
  note *conv* = *o-def image-mset-union image-empty image-mset-add-mset Un-empty-left*
    set-mset-add-mset-insert set-mset-union image-Un image-insert set-mset-empty
    set-mset-mset set-image-mset
    set-map image-comp insert-is-Un[*symmetric*]
  define P' **where** $P' = \{$*mp-mset* ‘ *set-mset x* |. $x \in$ *set-mset P*$\}$
  from $*$(*1*)
  have *insert* (*pat-mset p*) (*pats-mset P*) $\Rrightarrow_s$ *pats-mset ps* $\cup$ *pats-mset P*
    unfolding *conv* P'*-def*[*symmetric*]
  proof *induction*
    case (*pat-remove-pp p*)
    show *?case* **unfolding** *conv*
      by (*intro P-remove-pp pp-success.intros*)
  next
    case $*$: (*pat-simp-mp mp mp′ p*)
    from *P-simp*[*OF pp-simp-mp*[*OF mp-step-mset-mp-trans*[*OF* $*$]]]
    show *?case* **by** *auto*
  next
    case $*$: (*pat-remove-mp mp p*)
    from *P-simp*[*OF pp-remove-mp*[*OF match-fail-mp-fail*[*OF* $*$]]]
    show *?case* **by** *simp*
  next
    case $*$: (*pat-instantiate n mp p x l s y t*)
    from $*$(*2*) **have** $x \in$ *tvars-mp* (*mp-mset mp*)
      using *conflicts*(*4*)[*of s t x*] **unfolding** *tvars-mp-def*
      by (*auto intro*!:*term.set-intros*(*3*))
    hence *x*: $x \in$ *tvars-pp* (*pat-mset* (*add-mset mp p*)) **unfolding** *tvars-pp-def*
      using $*$(*2*) **by** *auto*
    show *?case* **unfolding** *conv* τ*s-list*
      apply (*rule P-step-set-cong*[*OF P-instantiate*[*OF* $*$(*1*) *x*]])

by (*unfold conv subst-defs set-map image-comp*, *auto*)
qed
thus *?case* **unfolding** *conv* **.**
qed

lemma *P-step-pp-trans*: **assumes** $(P,Q) \in \Rrightarrow$
shows *pats-mset P* $\Rrightarrow_s$ *pats-mset Q*
by (*rule P-step-mset-imp-set*, *insert assms*, *unfold P-step-def*, *auto*)

theorem *P-step-pcorrect*: **assumes** *wf*: *wf-pats* (*pats-mset P*) **and** *step*: $(P,Q) \in$ *P-step*
shows *wf-pats* (*pats-mset Q*) $\wedge$ (*pats-complete* (*pats-mset P*) = *pats-complete* (*pats-mset Q*))
proof −
note *step* = *P-step-pp-trans*[*OF step*]
from *P-step-set-pcorrect*[*OF step*] *P-step-set-wf*[*OF step*] *wf*
show *?thesis* **by** *auto*
qed

corollary *P-steps-pcorrect*: **assumes** *wf*: *wf-pats* (*pats-mset P*)
and *step*: $(P,Q) \in \Rrightarrow^*$
shows *wf-pats* (*pats-mset Q*) $\wedge$ (*pats-complete* (*pats-mset P*) $\longleftrightarrow$ *pats-complete* (*pats-mset Q*))
using *step* **by** *induct* (*insert wf P-step-pcorrect*, *auto*)

Gather all results for the multiset-based implementation: decision procedure on well-formed inputs (termination was proven before)

theorem *P-step*:
assumes *wf*: *wf-pats* (*pats-mset P*) **and** *NF*: $(P,Q) \in \Rrightarrow^!$
shows $Q = \{\#\} \wedge$ *pats-complete* (*pats-mset P*) — either the result is and input P is complete
$\vee$ $Q =$ *bottom-mset* $\wedge \neg$ *pats-complete* (*pats-mset P*) — or the result = bot and P is not complete
proof −
from *NF* **have** *steps*: $(P,Q) \in \Rrightarrow\hat{}*$ **and** *NF*: $Q \in$ *NF P-step* **by** *auto*
from *P-steps-pcorrect*[*OF wf steps*]
have *wf*: *wf-pats* (*pats-mset Q*) **and**
sound: *pats-complete* (*pats-mset P*) = *pats-complete* (*pats-mset Q*)
by *blast+*
from *P-step-NF*[*OF NF*] **have** $Q \in \{\{\#\},$*bottom-mset*$\}$ **.**
thus *?thesis* **unfolding** *sound* **by** *auto*
qed

end
end

5 Computing Nonempty and Infinite sorts

This theory provides two algorithms, which both take a description of a set of sorts with their constructors. The first algorithm computes the set of sorts that are nonempty, i.e., those sorts that are inhabited by ground terms; and the second algorithm computes the set of sorts that are infinite, i.e., where one can build arbitrary large ground terms.

theory *Compute-Nonempty-Infinite-Sorts*
  imports
    Sorted-Terms.Sorted-Terms
    LP-Duality.Minimum-Maximum
    Matrix.Utility
begin

5.1 Deciding the nonemptyness of all sorts under consideration

function *compute-nonempty-main* :: $'\tau$ *set* $\Rightarrow$ $(('f \times '\tau$ *list*$) \times '\tau)$ *list* $\Rightarrow$ $'\tau$ *set*
where
  compute-nonempty-main ne ls = (*let rem-ls* = *filter* (λ *f*. *snd f* $\notin$ *ne*) *ls in*
    case partition (λ ((-,*args*),-). *set args* $\subseteq$ *ne*) *rem-ls of*
      (*new*, *rem*) $\Rightarrow$ *if new* = [] *then ne else compute-nonempty-main* (*ne* $\cup$ *set* (*map snd new*)) *rem*)
  by *pat-completeness auto*

termination
proof (*relation measure* (*length o snd*), *goal-cases*)
  case (*2 ne ls rem-ls new rem*)
  have *length new* + *length rem* = *length rem-ls*
    using *2*(*2*) *sum-length-filter-compl*[*of* - *rem-ls*] **by** (*auto simp*: *o-def*)
  with *2*(*3*) **have** *length rem* < *length rem-ls* **by** (*cases new*, *auto*)
  also have ... $\leq$ *length ls* **using** *2*(*1*) **by** *auto*
  finally show *?case* **by** *simp*
qed *simp*

declare *compute-nonempty-main.simps*[*simp del*]

definition *compute-nonempty-sorts* :: $(('f \times '\tau$ *list*$) \times '\tau)$ *list* $\Rightarrow$ $'\tau$ *set* **where**
  compute-nonempty-sorts Cs = *compute-nonempty-main* {} *Cs*

lemma *compute-nonempty-sorts*:
  assumes *distinct* (*map fst Cs*)
  and *map-of Cs* = *C*
shows *compute-nonempty-sorts Cs* = $\{\tau.\ \exists\ t :: ('f,'v)term.\ t : \tau\ in\ \mathcal{T}(C,\emptyset)\}$ (**is** - = *?NE*)
proof –
  let *?TC* = $\mathcal{T}(C,(\emptyset :: 'v \Rightarrow \text{-}))$
  have *ne* $\subseteq$ *?NE* $\Longrightarrow$ *set ls* $\subseteq$ *set Cs* $\Longrightarrow$ *snd* ‘ (*set Cs* $-$ *set ls*) $\subseteq$ *ne* $\Longrightarrow$

compute-nonempty-main ne ls = *?NE* **for** *ne ls*
proof (*induct ne ls rule*: *compute-nonempty-main.induct*)
case (*1 ne ls*)
note *ne* = *1*(*2*)
define *rem-ls* **where** *rem-ls* = *filter* (λ *f*. *snd f* $\notin$ *ne*) *ls*
have *rem-ls*: *set rem-ls* $\subseteq$ *set Cs*
snd ‘ (*set Cs* − *set rem-ls*) $\subseteq$ *ne*
using *1*(*2*−) **by** (*auto simp*: *rem-ls-def*)
obtain *new rem* **where** *part*: *partition* (λ((*f*, *args*), *target*). *set args* $\subseteq$ *ne*) *rem-ls* = (*new*,*rem*) **by** *force*
have [*simp*]: *compute-nonempty-main ne ls* = (*if new* = [] *then ne else compute-nonempty-main* (*ne* $\cup$ *set* (*map snd new*)) *rem*)
unfolding *compute-nonempty-main.simps*[*of ne ls*] *Let-def rem-ls-def*[*symmetric*] *part* **by** *auto*
have *new*: *set* (*map snd new*) $\subseteq$ *?NE*
proof
fix τ
assume $\tau \in$ *set* (*map snd new*)
then obtain *f args* **where** ((*f*,*args*),τ) $\in$ *set rem-ls* **and** *args*: *set args* $\subseteq$ *ne* **using** *part* **by** *auto*
with *rem-ls* **have** ((*f*,*args*),τ) $\in$ *set Cs* **by** *auto*
with *assms* **have** *C* (*f*,*args*) = *Some* τ **by** *auto*
hence *fC*: *f* : *args* $\to$ τ *in C* **by** (*simp add*: *hastype-in-ssig-def*)
from *args ne* **have** $\forall$ *tau*. $\exists$ *t*. *tau* $\in$ *set args* $\longrightarrow$ *t* : *tau in ?TC* **by** *auto*
from *choice*[*OF this*] **obtain** *ts* **where** $\bigwedge$ *tau*. *tau* $\in$ *set args* $\Longrightarrow$ *ts tau* : *tau in ?TC* **by** *auto*
hence *Fun f* (*map ts args*) : τ *in ?TC*
apply (*intro Fun-hastypeI*[*OF fC*])
by (*simp add*: *list-all2-conv-all-nth*)
thus $\tau \in$ *?NE* **by** *auto*
qed
show *?case*
proof (*cases new* = [])
case *False*
note *IH* = *1*(*1*)[*OF rem-ls-def part*[*symmetric*] *False*]
have *compute-nonempty-main ne ls* = *compute-nonempty-main* (*ne* $\cup$ *set* (*map snd new*)) *rem* **using** *False* **by** *simp*
also have . . . = *?NE*
proof (*rule IH*)
show *ne* $\cup$ *set* (*map snd new*) $\subseteq$ *?NE* **using** *new ne* **by** *auto*
show *set rem* $\subseteq$ *set Cs* **using** *rem-ls part* **by** *auto*
show *snd* ‘ (*set Cs* − *set rem*) $\subseteq$ *ne* $\cup$ *set* (*map snd new*)
proof
fix τ
assume $\tau \in$ *snd* ‘ (*set Cs* − *set rem*)
then obtain *f args* **where** *in-ls*: ((*f*,*args*),τ) $\in$ *set Cs* **and** *nrem*: ((*f*,*args*),τ) $\notin$ *set rem* **by** *force*
thus $\tau \in$ *ne* $\cup$ *set* (*map snd new*) **using** *new part rem-ls* **by** *force*
qed

qed
finally show *?thesis* **.**
next
case *True*
have *compute-nonempty-main ne ls* = *ne* **using** *True* **by** *simp*
also have ... = *?NE*
proof (*rule ccontr*)
assume ¬ *?thesis*
with *ne* **obtain** τ *t* **where** *counter*: *t* : τ *in ?TC* $\tau \notin$ *ne* **by** *auto*
thus *False*
proof (*induct t* τ)
case (*Fun f ts* τs τ)
from *Fun*(*1*) **have** *C* (*f*,τs) = *Some* τ **by** (*simp add*: *hastype-in-ssig-def*)
with *assms*(*2*) **have** *mem*: ((*f*,τs),τ) $\in$ *set Cs* **by** (*meson map-of-SomeD*)
from *Fun*(*3*) **have** τs: *set* $\tau s \subseteq$ *ne* **by** (*induct*, *auto*)
from *rem-ls mem Fun*(*4*) **have** ((*f*,τs),τ) $\in$ *set rem-ls* **by** *auto*
with τs **have** ((*f*,τs),τ) $\in$ *set new* **using** *part* **by** *auto*
with *True* **show** *?case* **by** *auto*
qed *auto*
qed
finally show *?thesis* **.**
qed
qed
from *this*[*of* {} *Cs*] **show** *?thesis* **unfolding** *compute-nonempty-sorts-def* **by** *auto*
qed

definition *decide-nonempty-sorts* :: '*t list* ⇒ (('*f* × '*t list*) × '*t*)*list* ⇒ '*t option*
where
decide-nonempty-sorts τs *Cs* = (*let ne* = *compute-nonempty-sorts Cs in*
find (λ τ. $\tau \notin$ *ne*) τs)

lemma *decide-nonempty-sorts*:
assumes *distinct* (*map fst Cs*)
and *map-of Cs* = *C*
shows *decide-nonempty-sorts* τs *Cs* = *None* $\Longrightarrow \forall$ $\tau \in$ *set* τs. $\exists$ *t* :: ('*f*,'*v*)*term*. *t* : τ *in* $\mathcal{T}(C,\emptyset)$
decide-nonempty-sorts τs *Cs* = *Some* $\tau \Longrightarrow \tau \in$ *set* $\tau s \land \neg$ ($\exists$ *t* :: ('*f*,'*v*)*term*. *t* : τ *in* $\mathcal{T}(C,\emptyset)$)
unfolding *decide-nonempty-sorts-def Let-def compute-nonempty-sorts*[*OF assms*, **where** *?'v* = *'v*]
find-None-iff find-Some-iff **by** *auto*

5.2 Deciding infiniteness of a sort

We provide an algorithm, that given a list of sorts with constructors, computes the set of those sorts that are infinite. Here a sort is defined as infinite iff there is no upper bound on the size of the ground terms of that sort.

function *compute-inf-main* :: 'τ *set* ⇒ ('τ × ('*f* × 'τ *list*)*list*) *list* ⇒ 'τ *set* **where**

compute-inf-main m-inf ls = (
let (*fin*, *ls*′) =
partition (λ (τ,*fs*). ∀ τ*s* ∈ *set* (*map snd fs*). ∀ τ ∈ *set* τ*s*. τ ∉ *m-inf*) *ls*
in if fin = [] *then m-inf else compute-inf-main* (*m-inf* − *set* (*map fst fin*)) *ls*′)
by *pat-completeness auto*

termination
proof (*relation measure* (*length o snd*), *goal-cases*)
case (*2 m-inf ls pair fin ls*′)
have *length fin* + *length ls*′ = *length ls*
using *2 sum-length-filter-compl*[*of - ls*] **by** (*auto simp*: *o-def*)
with *2*(*3*) **have** *length ls*′ < *length ls* **by** (*cases fin*, *auto*)
thus *?case* **by** *auto*
qed *simp*

lemma *compute-inf-main*: **fixes** *E* :: ′*v* ⇀ ′*t* **and** *C* :: (′*f*,′*t*)*ssig*
assumes *E*: *E* = ∅
and *C-Cs*: *C* = *map-of Cs*′
and *Cs*′: *set Cs*′ = *set* (*concat* (*map* ((λ (τ, *fs*). *map* (λ *f*. (*f*,τ)) *fs*)) *Cs*))
and *arg-types-inhabitet*: ∀ *f* τ*s* τ τ′. *f* : τ*s* → τ *in C* ⟶ τ′ ∈ *set* τ*s* ⟶ (∃ *t*. *t* : τ′ *in* 𝒯(*C*,*E*))
and *dist*: *distinct* (*map fst Cs*) *distinct* (*map fst Cs*′)
and *inhabitet*: ∀ τ *fs*. (τ,*fs*) ∈ *set Cs* ⟶ *set fs* ≠ {}
and ∀ τ. τ ∉ *m-inf* ⟶ *bdd-above* (*size* ‘ {*t*. *t* : τ *in* 𝒯(*C*,*E*)})
and *set ls* ⊆ *set Cs*
and *fst* ‘ (*set Cs* − *set ls*) ∩ *m-inf* = {}
and *m-inf* ⊆ *fst* ‘ *set ls*
shows *compute-inf-main m-inf ls* = {τ. ¬ *bdd-above* (*size* ‘ {*t*. *t* : τ *in* 𝒯(*C*,*E*)})}

using *assms*(*8*−)
proof (*induct m-inf ls rule*: *compute-inf-main.induct*)
case (*1 m-inf ls*)
let *?fin* = λ τ. *bdd-above* (*size* ‘ {*t*. *t* : τ *in* 𝒯(*C*,*E*)})
define *crit* **where** *crit* = (λ (τ :: ′*t*,*fs* :: (′*f* × ′*t list*) *list*). ∀ τ*s* ∈ *set* (*map snd fs*). ∀ τ ∈ *set* τ*s*. τ ∉ *m-inf*)
define *S* **where** *S* τ′ = *size* ‘ {*t*. *t* : τ′ *in* 𝒯(*C*,*E*)} **for** τ′
define *M* **where** *M* τ′ = *Maximum* (*S* τ′) **for** τ′
define *M*′ **where** *M*′ σ*s* = *sum-list* (*map M* σ*s*) + (*1* + *length* σ*s*) **for** σ*s*
define *L* **where** *L* = [σ*s* . (τ,*cs*) <− *Cs*, (*f*,σ*s*) <− *cs*]
define *N* **where** *N* = *max-list* (*map M*′ *L*)
obtain *fin ls*′ **where** *part*: *partition crit ls* = (*fin*, *ls*′) **by** *force*
{
fix τ *cs*
assume *inCs*: (τ,*cs*) ∈ *set Cs*
have *nonempty*:∃ *t*. *t* : τ *in* 𝒯(*C*,*E*)
proof −
from *inhabitet*[*rule-format*, *OF inCs*] **obtain** *f* σ*s* **where** (*f*,σ*s*) ∈ *set cs* **by** (*cases cs*,*auto*)
with *inCs* **have** ((*f*,σ*s*),τ) ∈ *set Cs*′ **unfolding** *Cs*′ **by** *auto*

hence *fC*: $f : \sigma s \to \tau$ *in C* **using** *dist(2)* **unfolding** *C-Cs*
by (*meson hastype-in-ssig-def map-of-is-SomeI*)
hence $\forall\sigma.\ \exists\ t.\ \sigma \in$ *set* $\sigma s \longrightarrow t : \sigma$ *in* $\mathcal{T}(C,E)$ **using** *arg-types-inhabitet*[*rule-format*, *of f* σs τ] **by** *auto*
from *choice*[*OF this*] **obtain** *t* **where** $\sigma \in$ *set* $\sigma s \Longrightarrow t\ \sigma : \sigma$ *in* $\mathcal{T}(C,E)$ **for** σ **by** *auto*
hence *Fun f* (*map t* σs) : τ *in* $\mathcal{T}(C,E)$ **using** *list-all2-conv-all-nth*
apply (*intro Fun-hastypeI*[*OF fC*]) **by** (*simp add*: *list-all2-conv-all-nth*)
then show *?thesis* **by** *auto*
qed
} **note** *inhabited* = *this*
{
fix τ
assume *asm*: $\tau \in$ *fst* ' *set fin*
hence *?fin* τ
proof(*cases* $\tau \in$ *m-inf*)
case *True*
then obtain *fs* **where** *taufs*:(τ, *fs*) $\in$ *set fin* **using** *asm* **by** *auto*
{
fix τ' **and** *t* **and** *args*
assume $*$: $\tau' \in$ *set args args* $\in$ *snd* ' *set fs t* : τ' *in* $\mathcal{T}(C,E)$
from $*$ **have** $\tau' \notin$ *m-inf* **using** *taufs* **unfolding** *compute-inf-main.simps*[*of m-inf*]
using *crit-def part* **by** *fastforce*
hence *?fin* τ' **using** *crit-def part 1(2)* **by** *auto*
hence *hM*: *bdd-above* (*S* τ') **unfolding** *S-def* **.**
from $*$(*3*) **have** *size t* $\in$ *S* τ' **unfolding** *S-def* **by** *auto*
from *this hM* **have** *size t* $\leq$ *M* τ' **unfolding** *M-def* **by** (*metis bdd-above-Maximum-nat*)
} **note** *arg-type-bounds* = *this*
{
fix *t*
assume *t*: *t* : τ *in* $\mathcal{T}(C,E)$
then obtain *f ts* **where** *tF*: *t* = *Fun f ts* **unfolding** *E* **by** (*induct*, *auto*)
from *t*[*unfolded tF Fun-hastype*]
obtain σs **where** *f*: $f : \sigma s \to \tau$ *in C* **and** *args*: *ts* $:_l$ σs *in* $\mathcal{T}(C,E)$ **by** *auto*
from *part*[*simplified*] *asm 1(3)* **obtain** *cs* **where** *inCs*: (τ,*cs*) $\in$ *set Cs* **and** *crit*: *crit* (τ,*cs*) **by** *auto*
{
from *f*[*unfolded hastype-in-ssig-def C-Cs*]
have *map-of Cs'* (f, σs) = *Some* τ **by** *auto*
hence ((f,σs), τ) $\in$ *set Cs'* **by** (*metis map-of-SomeD*)
from *this*[*unfolded Cs'*, *simplified*] **obtain** *cs'* **where** *2*: (τ,*cs'*) $\in$ *set Cs* **and** *mem*: (f,σs) $\in$ *set cs'* **by** *auto*
from *inCs 2 dist* **have** *cs'* = *cs* **by** (*metis eq-key-imp-eq-value*)
with *mem* **have** *mem*: (f,σs) $\in$ *set cs* **by** *auto*
} **note** *mem* = *this*
from *mem inCs* **have** *inL*: $\sigma s \in$ *set L* **unfolding** *L-def* **by** *force*
{
fix σ *ti*

assume $\sigma \in$ *set* σs **and** *ti*: *ti* : σ *in* $\mathcal{T}(C,E)$
with *mem crit* **have** $\sigma \notin$ *m-inf* **unfolding** *crit-def* **by** *auto*
hence *?fin* σ **using** *1(2)* **by** *auto*
hence *hM*: *bdd-above* ($S\ \sigma$) **unfolding** *S-def* **.**
from *ti* **have** *size ti* $\in S\ \sigma$ **unfolding** *S-def* **by** *auto*
from *this hM* **have** *size ti* $\leq M\ \sigma$ **unfolding** *M-def* **by** (*metis bdd-above-Maximum-nat*)
} **note** *arg-bound* = *this*
have *len*: *length* σs = *length ts* **using** *args* **by** (*auto simp*: *list-all2-conv-all-nth*)
have *size t* = *sum-list* (*map size ts*) + (*1* + *length ts*) **unfolding** *tF* **by** (*simp add*: *size-list-conv-sum-list*)
also have ... $\leq$ *sum-list* (*map M* σs) + (*1* + *length ts*) **unfolding** *tF args*
proof −
have *id1*: *map size ts* = *map* (λ *i*. *size* (*ts* ! *i*)) [*0* ..< *length ts*] **by** (*intro nth-equalityI*, *auto*)
have *id2*: *map M* σs = *map* (λ *i*. *M* (σs ! *i*)) [*0* ..< *length ts*] **using** *len* **by** (*intro nth-equalityI*, *auto*)
have *sum-list* (*map size ts*) $\leq$ *sum-list* (*map M* σs) **unfolding** *id1 id2*
apply (*rule sum-list-mono*) **using** *arg-bound args*
by (*auto*, *simp add*: *list-all2-conv-all-nth*)
thus *?thesis* **by** *auto*
qed
also have ... = *sum-list* (*map M* σs) + (*1* + *length* σs) **using** *args* **unfolding** *M-def* **using** *list-all2-lengthD* **by** *auto*
also have ... = M' σs **unfolding** M'*-def* **by** *auto*
also have ... $\leq$ *max-list* (*map* M' *L*)
by (*rule max-list*, *insert inL*, *auto*)
also have ... = *N* **unfolding** *N-def* **..**
finally have *size t* $\leq N$ **.**
}
hence $\bigwedge$ *s*. $s \in S\ \tau \Longrightarrow s \leq N$ **unfolding** *S-def* **by** *auto*
hence *finite* ($S\ \tau$)
using *finite-nat-set-iff-bounded-le* **by** *auto*
moreover
have *nonempty*:$\exists$ *t*. *t* : τ *in* $\mathcal{T}(C,E)$
proof −
from *part*[*simplified*] *asm 1(3)* **obtain** *cs* **where** *inCs*: (τ,*cs*) $\in$ *set Cs* **by** *auto*
thus *?thesis* **using** *inhabited* **by** *auto*
qed
hence $S\ \tau \neq \{\}$ **unfolding** *S-def* **by** *auto*
ultimately show *?thesis* **unfolding** *S-def*[*symmetric*] **by** (*metis Max-ge bdd-above-def*)
next
case *False*
then show *?thesis* **using** *1(2)* **by** *simp*
qed
} **note** *fin* = *this*
show *?case*
proof (*cases fin* = [])

case *False*
hence *compute-inf-main m-inf ls* = *compute-inf-main* (*m-inf* − *set* (*map fst fin*)) *ls'*
unfolding *compute-inf-main.simps*[*of m-inf*] *part*[*unfolded crit-def*] **by** *auto*
also have ... = {τ. ¬ *?fin* τ}
proof (*rule 1*(*1*)[*OF refl part*[*unfolded crit-def*, *symmetric*] *False*])
show *set ls'* ⊆ *set Cs* **using** *1*(*3*) *part* **by** *auto*
show *fst* ‘ (*set Cs* − *set ls'*) ∩ (*m-inf* − *set* (*map fst fin*)) = {} **using** *1*(*3*−*4*) *part* **by** *force*
show ∀ τ. τ ∉ *m-inf* − *set* (*map fst fin*) ⟶ *?fin* τ **using** *1*(*2*) *fin* **by** *force*
show *m-inf* − *set* (*map fst fin*) ⊆ *fst* ‘ *set ls'* **using** *1*(*5*) *part* **by** *force*
qed
finally show *?thesis* **.**
next
case *True*
hence *compute-inf-main m-inf ls* = *m-inf*
unfolding *compute-inf-main.simps*[*of m-inf*] *part*[*unfolded crit-def*] **by** *auto*
also have ... = {τ. ¬ *?fin* τ}
proof
show {τ. ¬ *?fin* τ} ⊆ *m-inf* **using** *fin 1*(*2*) **by** *auto*
{
fix τ
assume τ ∈ *m-inf*
with *1*(*5*) **obtain** *cs* **where** *mem*: (τ,*cs*) ∈ *set ls* **by** *auto*
from *part True* **have** *ls'*: *ls'* = *ls* **by** (*induct ls arbitrary*: *ls'*, *auto*)
from *partition-P*[*OF part*, *unfolded ls'*]
have ⋀ *e*. *e* ∈ *set ls* ⟹ ¬ *crit e* **by** *auto*
from *this*[*OF mem*, *unfolded crit-def split*]
obtain *c* τs τ' **where** ∗: (*c*,τs) ∈ *set cs* τ' ∈ *set* τs τ' ∈ *m-inf* **by** *auto*
from *mem 1*(*2*−) **have** (τ,*cs*) ∈ *set Cs* **by** *auto*
with ∗ **have** ((*c*,τs),τ) ∈ *set Cs'* **unfolding** *Cs'* **by** *force*
with *dist*(*2*) **have** *map-of Cs'* ((*c*,τs)) = *Some* τ **by** *simp*
from *this*[*folded C-Cs*] **have** *c*: *c* : τs → τ *in C* **unfolding** *hastype-in-ssig-def* **.**
from *arg-types-inhabitet this* **have** ∀ σ. ∃ *t*. σ ∈ *set* τs ⟶ *t* : σ *in* $\mathcal{T}(C,E)$ **by** *auto*
from *choice*[*OF this*] **obtain** *t* **where** ⋀ σ. σ ∈ *set* τs ⟹ *t* σ : σ *in* $\mathcal{T}(C,E)$ **by** *auto*
hence *list*: *map t* τs $:_l$ τs *in* $\mathcal{T}(C,E)$ **by** (*simp add*: *list-all2-conv-all-nth*)
with *c* **have** *Fun c* (*map t* τs) : τ *in* $\mathcal{T}(C,E)$ **by** (*intro Fun-hastypeI*)
with ∗ *c list* **have** ∃ *c* τs τ' *ts*. *Fun c ts* : τ *in* $\mathcal{T}(C,E)$ ∧ *ts* $:_l$ τs *in* $\mathcal{T}(C,E)$ ∧ *c* : τs → τ *in C* ∧ τ' ∈ *set* τs ∧ τ' ∈ *m-inf*
by *blast*
} **note** *m-invD* = *this*
{
fix *n* :: *nat*
have τ ∈ *m-inf* ⟹ ∃ *t*. *t* : τ *in* $\mathcal{T}(C,E)$ ∧ *size t* ≥ *n* **for** τ
proof (*induct n arbitrary*: τ)
case (*0* τ)

from *m-invD*[*OF 0*] **show** *?case* **by** *blast*
next
case (*Suc n* τ)
from *m-invD*[*OF Suc*(*2*)] **obtain** *c* τs τ' *ts*
where $*$: *ts* $:_l$ τs *in* $\mathcal{T}(C,E)$ *c* : $\tau s \to \tau$ *in C* $\tau' \in$ *set* τs $\tau' \in$ *m-inf*
by *auto*
from $*$(*1*)[*unfolded list-all2-conv-all-nth*] $*$(*3*)[*unfolded set-conv-nth*]
obtain *i* **where** *i*: *i* < *length* τs **and** *tsi*:*ts* ! *i* : τ' *in* $\mathcal{T}(C,E)$ **and** *len*: *length ts* = *length* τs **by** *auto*
from *Suc*(*1*)[*OF* $*$(*4*)] **obtain** *t* **where** *t*:*t* : τ' *in* $\mathcal{T}(C,E)$ **and** *ns*:*n* $\leq$ *size t* **by** *auto*
define *ts'* **where** *ts'* = *ts*[*i* := *t*]
have *ts'* $:_l$ τs *in* $\mathcal{T}(C,E)$ **using** *list-all2-conv-all-nth* **unfolding** *ts'-def*
by (*metis* $*$(*1*) *tsi has-same-type i list-all2-update-cong list-update-same-conv t*(*1*))
hence $**$:*Fun c ts'* : τ *in* $\mathcal{T}(C,E)$ **apply** (*intro Fun-hastypeI*[*OF* $*$(*2*)]) **by** *fastforce*
have *t* $\in$ *set ts'* **unfolding** *ts'-def* **using** *t*
by (*simp add*: *i len set-update-memI*)
hence *size* (*Fun c ts'*) $\geq$ *Suc n* **using** $*$
by (*simp add*: *size-list-estimation' ns*)
thus *?case* **using** $**$ **by** *blast*
qed
} **note** *main* = *this*
show *m-inf* $\subseteq$ {τ. $\neg$ *?fin* τ}
proof (*standard*, *standard*)
fix τ
assume *asm*: $\tau \in$ *m-inf*
have $\exists$ *t*. *t* : τ *in* $\mathcal{T}(C,E)$ $\wedge$ *n* < *size t* **for** *n* **using** *main*[*OF asm*, *of Suc n*] **by** *auto*
thus $\neg$ *?fin* τ
by (*metis bdd-above-Maximum-nat imageI mem-Collect-eq order.strict-iff*)
qed
qed
finally show *?thesis* **.**
qed
qed

definition *compute-inf-sorts* :: ((*'f* $\times$ *'t list*) $\times$ *'t*)*list* $\Rightarrow$ *'t set* **where**
compute-inf-sorts Cs = (*let*
Cs' = *map* (λ τ. (τ, *map fst* (*filter*(λf. *snd f* = τ) *Cs*))) (*remdups* (*map snd Cs*))
in compute-inf-main (*set* (*map fst Cs'*)) *Cs'*)

lemma *compute-inf-sorts*:
fixes *E* :: *'v* $\rightharpoonup$ *'t* **and** *C* :: (*'f*,*'t*)*ssig*
assumes *E*: *E* = $\emptyset$
and *C-Cs*: *C* = *map-of Cs*
and *arg-types-inhabitet*: $\forall$ *f* τs τ τ'. *f* : $\tau s \to \tau$ *in C* $\longrightarrow$ $\tau' \in$ *set* τs $\longrightarrow$ ($\exists$ *t*.

$t : \tau'$ *in* $\mathcal{T}(C,E)$)
and *dist*: *distinct* (*map fst Cs*)
shows *compute-inf-sorts Cs* = $\{\tau.\ \neg$ *bdd-above* (*size* ‘ $\{t.\ t : \tau$ *in* $\mathcal{T}(C,E)\})\}$
proof −
define *taus* **where** *taus* = *remdups* (*map snd Cs*)
define *Cs'* **where** *Cs'* = *map* ($\lambda\ \tau$. (τ, *map fst* (*filter*(λf. *snd f* = τ) *Cs*))) *taus*
have *compute-inf-sorts Cs* = *compute-inf-main* (*set* (*map fst Cs'*)) *Cs'*
unfolding *compute-inf-sorts-def taus-def Cs'-def Let-def* **by** *auto*
also have ... = $\{\tau.\ \neg$ *bdd-above* (*size* ‘ $\{t.\ t : \tau$ *in* $\mathcal{T}(C,E)\})\}$
proof (*rule compute-inf-main*[*OF E C-Cs - arg-types-inhabitet - dist - - subset-refl*])
have *distinct taus* **unfolding** *taus-def* **by** *auto*
thus *distinct* (*map fst Cs'*) **unfolding** *Cs'-def map-map o-def fst-conv* **by** *auto*
show *set Cs* = *set* (*concat* (*map* ($\lambda(\tau$, *fs*). *map* (λf. (f, τ)) *fs*) *Cs'*))
unfolding *Cs'-def taus-def* **by** *force*
show $\forall\,\tau$ *fs*. (τ, *fs*) $\in$ *set Cs'* $\longrightarrow$ *set fs* $\neq$ {}
unfolding *Cs'-def taus-def* **by** (*force simp*: *filter-empty-conv*)
show *fst* ‘ (*set Cs'* − *set Cs'*) $\cap$ *set* (*map fst Cs'*) = {} **by** *auto*
show *set* (*map fst Cs'*) $\subseteq$ *fst* ‘ *set Cs'* **by** *auto*
show $\forall\,\tau.\ \tau \notin$ *set* (*map fst Cs'*) $\longrightarrow$ *bdd-above* (*size* ‘ $\{t.\ t : \tau$ *in* $\mathcal{T}(C,E)\}$)
proof (*intro allI impI*)
fix τ
assume $\tau \notin$ *set* (*map fst Cs'*)
hence $\tau \notin$ *snd* ‘ *set Cs* **unfolding** *Cs'-def taus-def* **by** *auto*
hence *diff*: *C f* $\neq$ *Some* τ **for** *f* **unfolding** *C-Cs*
by (*metis Some-eq-map-of-iff dist imageI snd-conv*)
{
fix *t*
assume $t : \tau$ *in* $\mathcal{T}(C,E)$
hence *False* **using** *diff* **unfolding** *E*
proof *induct*
case (*Fun f ss* σs τ)
from *Fun*(*1*,*4*) **show** *False* **unfolding** *hastype-in-ssig-def* **by** *auto*
qed *auto*
}
hence *id*: $\{t.\ t : \tau$ *in* $\mathcal{T}(C,E)\}$ = {} **by** *auto*
show *bdd-above* (*size* ‘ $\{t.\ t : \tau$ *in* $\mathcal{T}(C,E)\}$) **unfolding** *id* **by** *auto*
qed
qed
finally show *?thesis* **.**
qed
end

6 A List-Based Implementation to Decide Pattern Completeness

theory *Pattern-Completeness-List*
imports

Pattern-Completeness-Multiset
Compute-Nonempty-Infinite-Sorts
HOL−Library.AList
begin

6.1 Definition of Algorithm

We refine the non-deterministic multiset based implementation to a deterministic one which uses lists as underlying data-structure. For matching problems we distinguish several different shapes.

type-synonym $('a,'b)alist = ('a \times 'b)list$
type-synonym $('f,'v,'s)match\text{-}problem\text{-}list = (('f,nat \times 's)term \times ('f,'v)term)$ *list* — mp with arbitrary pairs
type-synonym $('f,'v,'s)match\text{-}problem\text{-}lx = ((nat \times 's) \times ('f,'v)term)\ list$ — mp where left components are variable
type-synonym $('f,'v,'s)match\text{-}problem\text{-}rx = ('v,('f,nat \times 's)term\ list)\ alist \times bool$ — mp where right components are variables
type-synonym $('f,'v,'s)match\text{-}problem\text{-}lr = ('f,'v,'s)match\text{-}problem\text{-}lx \times ('f,'v,'s)match\text{-}problem\text{-}rx$ — a partitioned mp
type-synonym $('f,'v,'s)pat\text{-}problem\text{-}list = ('f,'v,'s)match\text{-}problem\text{-}list\ list$
type-synonym $('f,'v,'s)pat\text{-}problem\text{-}lr = ('f,'v,'s)match\text{-}problem\text{-}lr\ list$
type-synonym $('f,'v,'s)pats\text{-}problem\text{-}list = ('f,'v,'s)pat\text{-}problem\text{-}list\ list$
type-synonym $('f,'v,'s)pat\text{-}problem\text{-}set\text{-}impl = (('f,nat \times 's)term \times ('f,'v)term)$ *list list*

abbreviation $mp\text{-}list :: ('f,'v,'s)match\text{-}problem\text{-}list \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}mset$

  where $mp\text{-}list \equiv mset$

abbreviation $mp\text{-}lx :: ('f,'v,'s)match\text{-}problem\text{-}lx \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}list$
  where $mp\text{-}lx \equiv map\ (map\text{-}prod\ Var\ id)$

definition $mp\text{-}rx :: ('f,'v,'s)match\text{-}problem\text{-}rx \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}mset$
  where $mp\text{-}rx\ mp = mset\ (List.maps\ (\lambda\ (x,ts).\ map\ (\lambda\ t.\ (t,Var\ x))\ ts)\ (fst\ mp))$

definition $mp\text{-}rx\text{-}list :: ('f,'v,'s)match\text{-}problem\text{-}rx \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}list$
  where $mp\text{-}rx\text{-}list\ mp = List.maps\ (\lambda\ (x,ts).\ map\ (\lambda\ t.\ (t,Var\ x))\ ts)\ (fst\ mp)$

definition $mp\text{-}lr :: ('f,'v,'s)match\text{-}problem\text{-}lr \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}mset$
  where $mp\text{-}lr\ pair = (case\ pair\ of\ (lx,rx) \Rightarrow mp\text{-}list\ (mp\text{-}lx\ lx) + mp\text{-}rx\ rx)$

definition $mp\text{-}lr\text{-}list :: ('f,'v,'s)match\text{-}problem\text{-}lr \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}list$
  where $mp\text{-}lr\text{-}list\ pair = (case\ pair\ of\ (lx,rx) \Rightarrow mp\text{-}lx\ lx\ @\ mp\text{-}rx\text{-}list\ rx)$

definition $pat\text{-}lr :: ('f,'v,'s)pat\text{-}problem\text{-}lr \Rightarrow ('f,'v,'s)pat\text{-}problem\text{-}mset$
  where $pat\text{-}lr\ ps = mset\ (map\ mp\text{-}lr\ ps)$

definition $pat\text{-}mset\text{-}list :: ('f,'v,'s)pat\text{-}problem\text{-}list \Rightarrow ('f,'v,'s)pat\text{-}problem\text{-}mset$

where *pat-mset-list ps* = *mset* (*map mp-list ps*)

definition *pat-list* :: ('*f*,'*v*,'*s*)*pat-problem-list* ⇒ ('*f*,'*v*,'*s*)*pat-problem-set*
where *pat-list ps* = *set* ' *set ps*

abbreviation *pats-mset-list* :: ('*f*,'*v*,'*s*)*pats-problem-list* ⇒ ('*f*,'*v*,'*s*)*pats-problem-mset*

where *pats-mset-list* ≡ *mset o map pat-mset-list*

definition *subst-match-problem-list* :: ('*f*,*nat* × '*s*)*subst* ⇒ ('*f*,'*v*,'*s*)*match-problem-list* ⇒ ('*f*,'*v*,'*s*)*match-problem-list* **where**
subst-match-problem-list τ = *map* (*subst-left* τ)

definition *subst-pat-problem-list* :: ('*f*,*nat* × '*s*)*subst* ⇒ ('*f*,'*v*,'*s*)*pat-problem-list* ⇒ ('*f*,'*v*,'*s*)*pat-problem-list* **where**
subst-pat-problem-list τ = *map* (*subst-match-problem-list* τ)

definition *match-var-impl* :: ('*f*,'*v*,'*s*)*match-problem-lr* ⇒ ('*f*,'*v*,'*s*)*match-problem-lr*
where
match-var-impl mp = (*case mp of* (*xl*,(*rx*,*b*)) ⇒
let xs = *remdups* (*List.maps* (*vars-term-list o snd*) *xl*)
in (*xl*,(*filter* (λ (*x*,*ts*). *tl ts* ≠ [] ∨ *x* ∈ *set xs*) *rx*),*b*))

definition *find-var* :: ('*f*,'*v*,'*s*)*match-problem-lr list* ⇒ - **where** *find-var p* = (*case concat* (*map* (λ (*lx*,-). *lx*) *p*) *of*
(*x*,*t*) # - ⇒ *x*
| [] ⇒ *let* (-,*rx*,*b*) = *hd* (*filter* (*Not o snd o snd*) *p*)
in case hd rx of (*x*, *s* # *t* # -) ⇒ *hd* (*the* (*conflicts s t*)))

definition *empty-lr* :: ('*f*,'*v*,'*s*)*match-problem-lr* ⇒ *bool* **where**
empty-lr mp = (*case mp of* (*lx*,*rx*,-) ⇒ *lx* = [] ∧ *rx* = [])

context *pattern-completeness-context*
begin

insert an element into the part of the mp that stores pairs of form (t,x) for variables x. Internally this is represented as maps (assoc lists) from x to terms t1,t2,... so that linear terms are easily identifiable. Duplicates will be removed and clashes will be immediately be detected and result in None.

definition *insert-rx* :: ('*f*,*nat* × '*s*)*term* ⇒ '*v* ⇒ ('*f*,'*v*,'*s*)*match-problem-rx* ⇒ ('*f*,'*v*,'*s*)*match-problem-rx option* **where**
insert-rx t x rxb = (*case rxb of* (*rx*,*b*) ⇒ (*case map-of rx x of*
None ⇒ *Some* (((*x*,[*t*]) # *rx*, *b*))
| *Some ts* ⇒ (*case those* (*map* (*conflicts t*) *ts*)
of None ⇒ *None* — clash
| *Some cs* ⇒ *if* [] ∈ *set cs then Some rxb* — empty conflict means (t,x) was already part of rxb
else Some ((*AList.update x* (*t* # *ts*) *rx*, *b* ∨ (∃ *y* ∈ *set* (*concat cs*). *inf-sort* (*snd y*))))

)))

lemma *size-zip*[*termination-simp*]: *length ts* = *length ls* $\Longrightarrow$ *size-list* (λ*p. size* (*snd p*)) (*zip ts ls*)
  < *Suc* (*size-list size ls*)
  by (*induct ts ls rule*: *list-induct2*, *auto*)

Decomposition applies decomposition, duplicate and clash rule to classify all remaining problems as being of kind (x,f(l1,..,ln)) or (t,x).

fun *decomp-impl* :: ('*f*,'*v*,'*s*)*match-problem-list* ⇒ ('*f*,'*v*,'*s*)*match-problem-lr option*
where
  decomp-impl [] = *Some* ([],([],*False*))
| *decomp-impl* ((*Fun f ts*, *Fun g ls*) # *mp*) = (*if* (*f*,*length ts*) = (*g*,*length ls*) *then*
    decomp-impl (*zip ts ls* @ *mp*) *else None*)
| *decomp-impl* ((*Var x*, *Fun g ls*) # *mp*) = (*case decomp-impl mp of Some* (*lx*,*rx*) ⇒ *Some* ((*x*,*Fun g ls*) # *lx*,*rx*)
    | *None* ⇒ *None*)
| *decomp-impl* ((*t*, *Var y*) # *mp*) = (*case decomp-impl mp of Some* (*lx*,*rx*) ⇒
    (*case insert-rx t y rx of Some rx'* ⇒ *Some* (*lx*,*rx'*) | *None* ⇒ *None*)
    | *None* ⇒ *None*)

definition *match-steps-impl* :: ('*f*,'*v*,'*s*)*match-problem-list* ⇒ ('*f*,'*v*,'*s*)*match-problem-lr option* **where**
  match-steps-impl mp = *map-option match-var-impl* (*decomp-impl mp*)

fun *pat-inner-impl* :: ('*f*,'*v*,'*s*)*pat-problem-list* ⇒ ('*f*,'*v*,'*s*)*pat-problem-lr* ⇒ ('*f*,'*v*,'*s*)*pat-problem-lr option* **where**
  pat-inner-impl [] *pd* = *Some pd*
| *pat-inner-impl* (*mp* # *p*) *pd* = (*case match-steps-impl mp of*
    None ⇒ *pat-inner-impl p pd*
  | *Some mp'* ⇒ *if empty-lr mp' then None*
    else pat-inner-impl p (*mp'* # *pd*))

definition *pat-impl* :: *nat* ⇒ ('*f*,'*v*,'*s*)*pat-problem-list* ⇒ ('*f*,'*v*,'*s*)*pat-problem-list list option* **where**
  pat-impl n p = (*case pat-inner-impl p* [] *of None* ⇒ *Some* []
    | *Some p'* ⇒ (*if* (∀ *mp* ∈ *set p'*. *snd* (*snd mp*)) *then None* — detected inf-var-conflict (or empty mp)
      else let p'l = *map mp-lr-list p'*;
        x = *find-var p'*
      in
        Some (*map* (λ τ. *subst-pat-problem-list* τ *p'l*) (τ*s-list n x*))))

partial-function (*tailrec*) *pats-impl* :: *nat* ⇒ ('*f*,'*v*,'*s*)*pats-problem-list* ⇒ *bool*
where
  pats-impl n ps = (*case ps of* [] ⇒ *True*
    | *p* # *ps1* ⇒ (*case pat-impl n p of*
      None ⇒ *False*
    | *Some ps2* ⇒ *pats-impl* (*n* + *m*) (*ps2* @ *ps1*)))

definition *pat-complete-impl* :: $('f,'v,'s)$*pats-problem-list* $\Rightarrow$ *bool* **where**
pat-complete-impl ps = (*let*
n = *Suc* (*max-list* (*List.maps* (*map fst o vars-term-list o fst*) (*concat* (*concat ps*))))
in pats-impl n ps)
end

lemmas *pat-complete-impl-code* =
pattern-completeness-context.pat-complete-impl-def
pattern-completeness-context.pats-impl.simps
pattern-completeness-context.pat-impl-def
pattern-completeness-context.τ*s-list-def*
pattern-completeness-context.insert-rx-def
pattern-completeness-context.decomp-impl.simps
pattern-completeness-context.match-steps-impl-def
pattern-completeness-context.pat-inner-impl.simps

declare *pat-complete-impl-code*[*code*]

6.2 Partial Correctness of the Implementation

We prove that the list-based implementation is a refinement of the multiset-based one.

lemma *mset-concat-union*:
mset (*concat xs*) = $\sum_{\#}$ (*mset* (*map mset xs*))
by (*induct xs*, *auto simp*: *union-commute*)

lemma *in-map-mset*[*intro*]:
$a \in\# A \Longrightarrow f\ a \in\#$ *image-mset f A*
unfolding *in-image-mset* **by** *simp*

lemma *mset-update*: *map-of xs x* = *Some y* $\Longrightarrow$
mset (*AList.update x z xs*) = (*mset xs* − {# (*x*,*y*) #}) + {# (*x*,*z*) #}
by (*induction xs*, *auto*)

lemma *set-update*: *map-of xs x* = *Some y* $\Longrightarrow$ *distinct* (*map fst xs*) $\Longrightarrow$
set (*AList.update x z xs*) = *insert* (*x*,*z*) (*set xs* − {(*x*,*y*)})
by (*induction xs*, *auto*)

context *pattern-completeness-context-with-assms*
begin

Various well-formed predicates for intermediate results

definition *wf-ts* :: $('f,$ *nat* $\times$ $'s)$ *term list* $\Rightarrow$ *bool* **where**
wf-ts ts = (*ts* $\neq$ [] $\wedge$ *distinct ts* $\wedge$ ($\forall$ *j* < *length ts*. $\forall$ *i* < *j*. *conflicts* (*ts* ! *i*) (*ts* ! *j*) $\neq$ *None*))

definition *wf-ts2* :: $('f, nat \times 's)$ *term list* $\Rightarrow$ *bool* **where**
wf-ts2 ts = (*length ts* $\geq$ *2* $\wedge$ *distinct ts* $\wedge$ ($\forall$ *j* < *length ts*. $\forall$ *i* < *j*. *conflicts* (*ts* ! *i*) (*ts* ! *j*) $\neq$ *None*))

definition *wf-lx* :: $('f,'v,'s)$*match-problem-lx* $\Rightarrow$ *bool* **where**
wf-lx lx = (*Ball* (*snd* ' *set lx*) *is-Fun*)

definition *wf-rx* :: $('f,'v,'s)$*match-problem-rx* $\Rightarrow$ *bool* **where**
wf-rx rx = (*distinct* (*map fst* (*fst rx*)) $\wedge$ (*Ball* (*snd* ' *set* (*fst rx*)) *wf-ts*) $\wedge$ *snd rx* = *inf-var-conflict* (*set-mset* (*mp-rx rx*)))

definition *wf-rx2* :: $('f,'v,'s)$*match-problem-rx* $\Rightarrow$ *bool* **where**
wf-rx2 rx = (*distinct* (*map fst* (*fst rx*)) $\wedge$ (*Ball* (*snd* ' *set* (*fst rx*)) *wf-ts2*) $\wedge$ *snd rx* = *inf-var-conflict* (*set-mset* (*mp-rx rx*)))

definition *wf-lr* :: $('f,'v,'s)$*match-problem-lr* $\Rightarrow$ *bool*
where *wf-lr pair* = (*case pair of* (*lx*,*rx*) $\Rightarrow$ *wf-lx lx* $\wedge$ *wf-rx rx*)

definition *wf-lr2* :: $('f,'v,'s)$*match-problem-lr* $\Rightarrow$ *bool*
where *wf-lr2 pair* = (*case pair of* (*lx*,*rx*) $\Rightarrow$ *wf-lx lx* $\wedge$ (*if lx* = [] *then wf-rx2 rx else wf-rx rx*))

definition *wf-pat-lr* :: $('f,'v,'s)$*pat-problem-lr* $\Rightarrow$ *bool* **where**
wf-pat-lr mps = (*Ball* (*set mps*) (λ *mp*. *wf-lr2 mp* $\wedge$ $\neg$ *empty-lr mp*))

lemma *mp-step-mset-cong*:
assumes $(\rightarrow_m)^{**}$ *mp mp'*
shows (*add-mset* (*add-mset mp p*) *P*, *add-mset* (*add-mset mp' p*) *P*) $\in \Rrightarrow^*$
using *assms*
proof *induct*
case (*step mp' mp''*)
from *P-simp-pp*[*OF pat-simp-mp*[*OF step*(*2*), *of p*], *of P*]
have (*add-mset* (*add-mset mp' p*) *P*, *add-mset* (*add-mset mp'' p*) *P*) $\in$ *P-step*
unfolding *P-step-def* **by** *auto*
with *step*(*3*)
show *?case* **by** *simp*
qed *auto*

lemma *mp-step-mset-vars*: **assumes** *mp* $\rightarrow_m$ *mp'*
shows *tvars-mp* (*mp-mset mp*) $\supseteq$ *tvars-mp* (*mp-mset mp'*)
using *assms* **by** *induct* (*auto simp*: *tvars-mp-def set-zip*)

lemma *mp-step-mset-steps-vars*: **assumes** $(\rightarrow_m)^{**}$ *mp mp'*
shows *tvars-mp* (*mp-mset mp*) $\supseteq$ *tvars-mp* (*mp-mset mp'*)
using *assms* **by** (*induct*, *insert mp-step-mset-vars*, *auto*)

Continue with properties of the sub-algorithms

lemma *insert-rx*: **assumes** *res*: *insert-rx t x rxb* = *res*

and *wf*: *wf-rx rxb*
and *mp*: *mp* = (*ls*,*rxb*)
shows *res* = *Some rx'* $\Longrightarrow$ $(\rightarrow_m)^{**}$ (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*)) (*mp-lr* (*ls*,*rx'*) + *M*) $\wedge$ *wf-rx rx'*
res = *None* $\Longrightarrow$ *match-fail* (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*))
proof −
obtain *rx b* **where** *rxb*: *rxb* = (*rx*,*b*) **by** *force*
note [*simp*] = *List.maps-def*
note *res* = *res*[*unfolded insert-rx-def*]
{
assume $*$: *res* = *None*
with *res rxb* **obtain** *ts* **where** *look*: *map-of rx x* = *Some ts* **by** (*auto split*: *option.splits*)
with *res*[*unfolded look Let-def rxb split*] $*$ **obtain** *t'* **where** *t'*: *t'* $\in$ *set ts* **and** *clash*: *Conflict-Clash t t'*
by (*auto split*: *if-splits option.splits*)
from *map-of-SomeD*[*OF look*] *t'* **have** (*t'*, *Var x*) $\in\#$ *mp-rx rxb*
unfolding *mp-rx-def rxb* **by** *auto*
hence (*t'*, *Var x*) $\in\#$ *mp-lr mp* + *M* **unfolding** *mp mp-lr-def* **by** *auto*
then obtain *mp'* **where** *mp*: *mp-lr mp* + *M* = *add-mset* (*t'*, *Var x*) *mp'* **by** (*rule mset-add*)
show *match-fail* (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*)) **unfolding** *mp*
by (*rule match-clash'*[*OF clash*])
}
{
assume *res* = *Some rx'*
note *res* = *res*[*unfolded this rxb split*]
show *mp-step-mset*^** (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*)) (*mp-lr* (*ls*,*rx'*) + *M*) $\wedge$ *wf-rx rx'*
proof (*cases map-of rx x*)
case *look*: *None*
from *res*[*unfolded this*]
have *rx'*: *rx'* = ((*x*,[*t*]) # *rx*, *b*) **by** *auto*
have *id*: *mp-rx rx'* = *add-mset* (*t*, *Var x*) (*mp-rx rxb*)
using *look* **unfolding** *mp-rx-def mset-concat-union mset-map rx' o-def rxb*
by *auto*
have [*simp*]: (*x*, *t*) $\notin$ *set rx* **for** *t* **using** *look*
using *weak-map-of-SomeI* **by** *force*
have *inf-var-conflict* (*mp-mset* (*mp-rx* ((*x*, [*t*]) # *rx*, *b*))) = *inf-var-conflict* (*mp-mset* (*mp-rx* (*rx*, *b*)))
unfolding *mp-rx-def fst-conv inf-var-conflict-def*
by (*intro ex-cong1*, *auto*)
hence *wf*: *wf-rx rx'* **using** *wf look* **unfolding** *wf-rx-def rx' rxb* **by** (*auto simp*: *wf-ts-def*)
show *?thesis* **unfolding** *mp mp-lr-def split id*
using *wf* **unfolding** *rx'* **by** *auto*
next
case *look*: (*Some ts*)
from *map-of-SomeD*[*OF look*] **have** *mem*: (*x*,*ts*) $\in$ *set rx* **by** *auto*

note *res* = *res*[*unfolded look option.simps Let-def*]
from *res* **obtain** *cs* **where** *those*: *those* (*map* (*conflicts t*) *ts*) = *Some cs* **by** (*auto split*: *option.splits*)
note *res* = *res*[*unfolded those option.simps*]
from *arg-cong*[*OF those*[*unfolded those-eq-Some*], *of set*] **have** *confl*: *conflicts t ‘ set ts* = *Some ‘ set cs* **by** *auto*
show *?thesis*
proof (*cases* [] ∈ *set cs*)
case *True*
with *res* **have** *rx′*: *rx′* = *rxb* **by** (*auto split*: *if-splits simp*: *mp rxb those*)
from *True confl* **obtain** *t′* **where** *t′* ∈ *set ts* **and** *conflicts t t′* = *Some* [] **by** *force*
hence *t*: *t* ∈ *set ts* **using** *conflicts*(*5*)[*of t t′*] **by** *auto*
hence (*t*, *Var x*) ∈# *mp-rx rxb* **unfolding** *mp-rx-def rxb* **using** *mem* **by** *auto*
hence (*t*, *Var x*) ∈# *mp-lr mp* + *M* **unfolding** *mp mp-lr-def* **by** *auto*
then obtain *sub* **where** *id*: *mp-lr mp* + *M* = *add-mset* (*t*, *Var x*) *sub* **by** (*rule mset-add*)
show *?thesis* **unfolding** *id rx′ mp*[*symmetric*] **using** *match-duplicate*[*of* (*t*, *Var x*) *sub*] *wf* **by** *auto*
next
case *False*
with *res* **have** *rx′*: *rx′* = (*AList.update x* (*t* # *ts*) *rx*, *b* ∨ (∃ *y*∈*set* (*concat cs*). *inf-sort* (*snd y*))) **by** (*auto split*: *if-splits*)
from *split-list*[*OF mem*] **obtain** *rx1 rx2* **where** *rx*: *rx* = *rx1* @ (*x*,*ts*) # *rx2* **by** *auto*
have *id*: *mp-rx rx′* = *add-mset* (*t*, *Var x*) (*mp-rx rxb*)
unfolding *rx′ mp-rx-def rxb* **by** (*simp add*: *mset-update*[*OF look*] *mset-concat-union*, *auto simp*: *rx*)
from *wf*[*unfolded wf-rx-def*] *rx rxb* **have** *ts*: *wf-ts ts* **and** *b*: *b* = *inf-var-conflict* (*mp-mset* (*mp-rx rxb*)) **by** *auto*
from *False confl conflicts*(*5*)[*of t t*] **have** *t*: *t* ∉ *set ts* **by** *force*
from *confl* **have** *None* ∉ *set* (*map* (*conflicts t*) *ts*) **by** *auto*
with *ts t* **have** *ts′*: *wf-ts* (*t* # *ts*) **unfolding** *wf-ts-def*
apply *clarsimp*
subgoal for *j i* **by** (*cases j*, *force*, *cases i*; *force simp*: *set-conv-nth*)
done
have *b*: (*b* ∨ (∃ *y*∈*set* (*concat cs*). *inf-sort* (*snd y*))) = *inf-var-conflict* (*mp-mset* (*add-mset* (*t*, *Var x*) (*mp-rx rxb*))) (**is** - = *?ivc*)
proof (*standard*, *elim disjE bexE*)
show *b* ⟹ *?ivc* **unfolding** *b inf-var-conflict-def* **by** *force*
{
fix *y*
assume *y*: *y* ∈ *set* (*concat cs*) **and** *inf*: *inf-sort* (*snd y*)
from *y confl* **obtain** *t′ ys* **where** *t′*: *t′* ∈ *set ts* **and** *c*: *conflicts t t′* = *Some ys* **and** *y*: *y* ∈ *set ys* **unfolding** *set-concat*
by (*smt* (*verit*, *del-insts*) *UnionE image-iff*)
have *y*: *Conflict-Var t t′ y* **using** *c y* **by** *auto*
from *mem t′* **have** (*t′*,*Var x*) ∈# *mp-rx rxb* **unfolding** *rxb mp-rx-def*

by *auto*
thus *?ivc* **unfolding** *inf-var-conflict-def* **using** *inf y* **by** *fastforce*
}
assume *?ivc*
from *this*[*unfolded inf-var-conflict-def*]
obtain *s1 s2 x′ y*
where *ic*: (*s1*, *Var x′*) ∈# *add-mset* (*t*, *Var x*) (*mp-rx rxb*) ∧ (*s2*, *Var x′*) ∈# *add-mset* (*t*, *Var x*) (*mp-rx rxb*) ∧ *Conflict-Var s1 s2 y* ∧ *inf-sort* (*snd y*)
by *blast*
show *b* ∨ (∃ *y*∈*set* (*concat cs*). *inf-sort* (*snd y*))
proof (*cases* (*s1*, *Var x′*) ∈# *mp-rx rxb* ∧ (*s2*, *Var x′*) ∈# *mp-rx rxb*)
case *True*
with *ic* **have** *b* **unfolding** *b inf-var-conflict-def* **by** *blast*
thus *?thesis* **..**
next
case *False*
with *ic* **have** (*s1*,*Var x′*) = (*t*,*Var x*) ∨ (*s2*,*Var x′*) = (*t*,*Var x*) **by** *auto*
hence ∃ *s y*. (*s*, *Var x*) ∈# *add-mset* (*t*, *Var x*) (*mp-rx rxb*) ∧ *Conflict-Var t s y* ∧ *inf-sort* (*snd y*)
proof
assume (*s1*, *Var x′*) = (*t*, *Var x*)
thus *?thesis* **using** *ic* **by** *blast*
next
assume ∗: (*s2*, *Var x′*) = (*t*, *Var x*)
with *ic* **have** *Conflict-Var s1 t y* **by** *auto*
hence *Conflict-Var t s1 y* **using** *conflicts-sym*[*of s1 t*] **by** (*cases conflicts s1 t*; *cases conflicts t s1*, *auto*)
with *ic* ∗ **show** *?thesis* **by** *blast*
qed
then obtain *s y* **where** *sx*: (*s*, *Var x*) ∈# *add-mset* (*t*, *Var x*) (*mp-rx rxb*) **and** *y*: *Conflict-Var t s y* **and** *inf*: *inf-sort* (*snd y*)
by *blast*
from *wf* **have** *dist*: *distinct* (*map fst rx*) **unfolding** *wf-rx-def rxb* **by** *auto*
from *y* **have** *s* ≠ *t* **by** *auto*
with *sx* **have** (*s*, *Var x*) ∈# *mp-rx rxb* **by** *auto*
hence *s* ∈ *set ts* **unfolding** *mp-rx-def rxb* **using** *mem eq-key-imp-eq-value*[*OF dist*] **by** *auto*
with *y confl* **have** *y* ∈ *set* (*concat cs*) **by** (*cases conflicts t s*; *force*)
with *inf* **show** *?thesis* **by** *auto*
qed
qed
have *wf*: *wf-rx rx′* **using** *wf ts′* **unfolding** *wf-rx-def id* **unfolding** *rx′ rxb snd-conv b* **by** (*auto simp*: *distinct-update set-update*[*OF look*])
show *?thesis* **using** *wf id* **unfolding** *mp* **by** (*auto simp*: *mp-lr-def*)
qed
qed
}
qed

lemma *decomp-impl*: *decomp-impl mp* = *res* $\Longrightarrow$
(*res* = *Some mp'* $\longrightarrow$ $(\to_m)^{**}$ (*mp-list mp* + *M*) (*mp-lr mp'* + *M*) $\wedge$ *wf-lr mp'*)
$\wedge$ (*res* = *None* $\longrightarrow$ ($\exists$ *mp'*. $(\to_m)^{**}$ (*mp-list mp* + *M*) *mp'* $\wedge$ *match-fail mp'*))
proof (*induct mp arbitrary*: *res M mp' rule*: *decomp-impl.induct*)
case *1*
thus *?case* **by** (*auto simp*: *mp-lr-def mp-rx-def List.maps-def wf-lr-def wf-lx-def wf-rx-def inf-var-conflict-def*)
next
case (*2 f ts g ls mp res M mp'*)
have *id*: *mp-list* ((*Fun f ts*, *Fun g ls*) # *mp*) + *M* = *add-mset* (*Fun f ts*, *Fun g ls*) (*mp-list mp* + *M*)
by *auto*
show *?case*
proof (*cases* (*f*,*length ts*) = (*g*,*length ls*))
case *False*
with *2*(*2*−) **have** *res*: *res* = *None* **by** *auto*
from *match-clash*[*OF False*, *of* (*mp-list mp* + *M*), *folded id*]
show *?thesis* **unfolding** *res* **by** *blast*
next
case *True*
have *id2*: *mp-list* (*zip ts ls* @ *mp*) + *M* = *mp-list mp* + *M* + *mp-list* (*zip ts ls*)
by *auto*
from *True 2*(*2*−) **have** *res*: *decomp-impl* (*zip ts ls* @ *mp*) = *res* **by** *auto*
note *IH* = *2*(*1*)[*OF True this*, *of mp' M*]
note *step* = *match-decompose*[*OF True*, *of mp-list mp* + *M*, *folded id id2*]
from *IH step* **show** *?thesis* **by** (*meson converse-rtranclp-into-rtranclp*)
qed
next
case (*3 x g ls mp res M mp'*)
note *res* = *3*(*2*)[*unfolded decomp-impl.simps*]
show *?case*
proof (*cases decomp-impl mp*)
case *None*
from *3*(*1*)[*OF None*, *of mp' add-mset* (*Var x*, *Fun g ls*) *M*] *None res* **show** *?thesis* **by** *auto*
next
case (*Some mpx*)
then obtain *lx rx* **where** *decomp*: *decomp-impl mp* = *Some* (*lx*,*rx*) **by** (*cases mpx*, *auto*)
from *res*[*unfolded decomp option.simps split*] **have** *res*: *res* = *Some* ((*x*, *Fun g ls*) # *lx*, *rx*) **by** *auto*
from *3*(*1*)[*OF decomp*, *of* (*lx*, *rx*) *add-mset* (*Var x*, *Fun g ls*) *M*] *res*
show *?thesis* **by** (*auto simp*: *mp-lr-def wf-lr-def wf-lx-def*)
qed
next
case (*4 t y mp res M mp'*)
note *res* = *4*(*2*)[*unfolded decomp-impl.simps*]

show *?case*
proof (*cases decomp-impl mp*)
case *None*
from *4(1)*[*OF None*, *of mp′ add-mset* (*t*, *Var y*) *M*] *None res* **show** *?thesis* **by** *auto*
next
case (*Some mpx*)
then obtain *lx rx* **where** *decomp*: *decomp-impl mp* = *Some* (*lx*,*rx*) **by** (*cases mpx*, *auto*)
note *res* = *res*[*unfolded decomp option.simps split*]
from *4(1)*[*OF decomp*, *of* (*lx*, *rx*) *add-mset* (*t*, *Var y*) *M*]
have *IH*: $(\to_m)^{**}$ (*mp-list* ((*t*, *Var y*) # *mp*) + *M*) (*mp-lr* (*lx*, *rx*) + *add-mset* (*t*, *Var y*) *M*)
wf-lr (*lx*, *rx*) **by** *auto*
from *IH* **have** *wf-rx*: *wf-rx rx* **unfolding** *wf-lr-def* **by** *auto*
show *?thesis*
proof (*cases insert-rx t y rx*)
case *None*
with *res* **have** *res*: *res* = *None* **by** *auto*
from *insert-rx*(*2*)[*OF None wf-rx refl refl*, *of lx M*]
IH res **show** *?thesis* **by** *auto*
next
case (*Some rx′*)
with *res* **have** *res*: *res* = *Some* (*lx*, *rx′*) **by** *auto*
from *insert-rx*(*1*)[*OF Some wf-rx refl refl*, *of lx M*]
have *wf-rx*: *wf-rx rx′*
and *steps*: $(\to_m)^{**}$ (*mp-lr* (*lx*, *rx*) + *add-mset* (*t*, *Var y*) *M*) (*mp-lr* (*lx*, *rx′*) + *M*)
by *auto*
from *IH*(*1*) *steps*
have *steps*: $(\to_m)^{**}$ (*mp-list* ((*t*, *Var y*) # *mp*) + *M*) (*mp-lr* (*lx*, *rx′*) + *M*) **by** *auto*
from *wf-rx IH*(*2*−) **have** *wf*: *wf-lr* (*lx*, *rx′*)
unfolding *wf-lr-def* **by** *auto*
from *res wf steps* **show** *?thesis* **by** *auto*
qed
qed
qed

lemma *match-var-impl*: **assumes** *wf*: *wf-lr mp*
shows $(\to_m)^{**}$ (*mp-lr mp*) (*mp-lr* (*match-var-impl mp*))
and *wf-lr2* (*match-var-impl mp*)
proof −
note [*simp*] = *List.maps-def*
let *?mp′* = *match-var-impl mp*
from *assms* **obtain** *xl rx b* **where** *mp3*: *mp* = (*xl*,(*rx*,*b*)) **by** (*cases mp*, *auto*)
define *xs* **where** *xs* = *remdups* (*List.maps* (*vars-term-list o snd*) *xl*)
have *xs*: *xl* = [] ⟹ *xs* = [] **unfolding** *xs-def* **by** *auto*
define *f* **where** *f* = (λ (*x*,*ts* :: (′*f*, *nat* × ′*s*)*term list*). *tl ts* ≠ [] ∨ *x* ∈ *set xs*)

define *mp′* **where** *mp′* = *mp-rx* (*filter f rx*, *b*) + *mp-list* (*mp-lx xl*)
define *deleted* **where** *deleted* = *mp-rx* (*filter* (*Not o f*) *rx*, *b*)
have *mp′*: *mp-lr ?mp′* = *mp′ ?mp′* = (*xl*, (*filter f rx*,*b*))
unfolding *mp3 mp′-def match-var-impl-def split xs-def f-def mp-lr-def* **by** *auto*
have *mp-rx* (*rx*,*b*) = *mp-rx* (*filter f rx*, *b*) + *mp-rx* (*filter* (*Not o f*) *rx*, *b*)
unfolding *mp-rx-def List.maps-def* **by** (*induct rx*, *auto*)
hence *mp*: *mp-lr mp* = *deleted* + *mp′* **unfolding** *mp3 mp-lr-def mp′-def deleted-def* **by** *auto*
have *inf-var-conflict* (*mp-mset* (*mp-rx* (*filter f rx*, *b*))) = *inf-var-conflict* (*mp-mset* (*mp-rx* (*rx*, *b*))) (**is** *?ivcf* = *?ivc*)
proof
show *?ivcf* $\Longrightarrow$ *?ivc* **unfolding** *inf-var-conflict-def mp-rx-def fst-conv List.maps-def* **by** *force*
assume *?ivc*
from *this*[*unfolded inf-var-conflict-def*]
obtain *s t x y* **where** *s*: (*s*, *Var x*) ∈# *mp-rx* (*rx*, *b*) **and** *t*: (*t*, *Var x*) ∈# *mp-rx* (*rx*, *b*) **and** *c*: *Conflict-Var s t y* **and** *inf*: *inf-sort* (*snd y*)
by *blast*
from *c conflicts*(*5*)[*of s t*] **have** *st*: *s* ≠ *t* **by** *auto*
from *s*[*unfolded mp-rx-def List.maps-def*]
obtain *ss* **where** *xss*: (*x*,*ss*) ∈ *set rx* **and** *s*: *s* ∈ *set ss* **by** *auto*
from *t*[*unfolded mp-rx-def List.maps-def*]
obtain *ts* **where** *xts*: (*x*,*ts*) ∈ *set rx* **and** *t*: *t* ∈ *set ts* **by** *auto*
from *wf*[*unfolded mp3 wf-lr-def wf-rx-def*] **have** *distinct* (*map fst rx*) **by** *auto*
from *eq-key-imp-eq-value*[*OF this xss xts*] *t* **have** *t*: *t* ∈ *set ss* **by** *auto*
with *s st* **have** *f* (*x*,*ss*) **unfolding** *f-def* **by** (*cases ss*; *cases tl ss*; *auto*)
hence (*x*, *ss*) ∈ *set* (*filter f rx*) **using** *xss* **by** *auto*
with *s t* **have** (*s*, *Var x*) ∈# *mp-rx* (*filter f rx*, *b*) (*t*, *Var x*) ∈# *mp-rx* (*filter f rx*, *b*)
unfolding *mp-rx-def List.maps-def* **by** *auto*
with *c inf*
show *?ivcf* **unfolding** *inf-var-conflict-def* **by** *blast*
qed
also have . . . = *b* **using** *wf* **unfolding** *mp3 wf-lr-def wf-rx-def* **by** *auto*
finally have *ivcf*: *?ivcf* = *b* **.**
show *wf-lr2* (*match-var-impl mp*)
proof (*cases xl* = [])
case *False*
from *ivcf False wf*[*unfolded mp3*] **show** *?thesis*
unfolding *mp′ wf-lr2-def wf-lr-def split wf-rx-def* **by** (*auto simp*: *distinct-map-filter*)
next
case *True*
with *xs* **have** *xs* = [] **by** *auto*
with *True wf*[*unfolded mp3*]
show *?thesis*
unfolding *wf-lr2-def mp′ split wf-rx2-def wf-rx-def ivcf*
unfolding *mp′ wf-lr2-def wf-lr-def split wf-rx-def wf-rx2-def wf-ts-def wf-ts2-def f-def*
apply (*clarsimp simp*: *distinct-map-filter*)

subgoal for *x ts* **by** (*cases ts*; *cases tl ts*; *force*)
done
qed
{
fix *xt t*
assume *del*: (*t*, *xt*) ∈# *deleted*
from *this*[*unfolded deleted-def mp-rx-def*, *simplified*]
obtain *x ts* **where** *mem*: (*x*,*ts*) ∈ *set rx* **and** *nf*: ¬ *f* (*x*, *ts*) **and** *t*: *t* ∈ *set ts* **and** *xt*: *xt* = *Var x* **by** *force*
note *del* = *del*[*unfolded xt*]
from *nf*[*unfolded f-def split*] *t* **have** *xxs*: *x* ∉ *set xs* **and** *ts*: *ts* = [*t*] **by** (*cases ts*; *cases tl ts*, *auto*)+
from *split-list*[*OF mem*[*unfolded ts*]] **obtain** *rx1 rx2* **where** *rx*: *rx* = *rx1* @ (*x*,[*t*]) # *rx2* **by** *auto*
from *wf*[*unfolded wf-lr-def mp3*] **have** *wf*: *wf-rx* (*rx*,*b*) **by** *auto*
hence *distinct* (*map fst rx*) **unfolding** *wf-rx-def* **by** *auto*
with *rx* **have** *xrx*: *x* ∉ *fst* ‘ *set rx1* ∪ *fst* ‘ *set rx2* **by** *auto*
define *mp′′* **where** *mp′′* = *mp-rx* (*filter* (*Not* ∘ *f*) (*rx1* @ *rx2*), *b*)
have *eq*: *deleted* = *add-mset* (*t*, *Var x*) *mp′′*
unfolding *deleted-def mp′′-def rx mp-rx-def List.maps-def mset-concat-union* **using** *nf ts* **by** *auto*
have ∃ *x mp′′*. *xt* = *Var x* ∧ *deleted* = *add-mset* (*t*, *Var x*) *mp′′* ∧ *x* ∉ ⋃ (*vars* ‘ *snd* ‘ (*mp-mset mp′′* ∪ *mp-mset mp′*))
proof (*intro exI conjI*, *rule xt*, *rule eq*, *intro notI*)
assume *x* ∈ ⋃ (*vars* ‘ *snd* ‘ (*mp-mset mp′′* ∪ *mp-mset mp′*))
then obtain *s t′* **where** *st*: (*s*,*t′*) ∈ *mp-mset* (*mp′* + *mp′′*) **and** *xt*: *x* ∈ *vars t′* **by** *force*
from *xrx* **have** (*s*,*t′*) ∉ *mp-mset mp′′* **using** *xt* **unfolding** *mp′′-def mp-rx-def* **by** *force*
with *st* **have** (*s*,*t′*) ∈ *mp-mset mp′* **by** *auto*
with *xxs* **have** (*s*, *t′*) ∈# *mp-rx* (*filter f rx*, *b*) **using** *xt* **unfolding** *xs-def mp′-def mp-rx-def*
by *auto*
with *xt nf* **show** *False* **unfolding** *mp-rx-def f-def split ts list.sel*
by *auto* (*metis Un-iff* ‹¬ (*tl ts* ≠ [] ∨ *x* ∈ *set xs*)› *fst-conv image-eqI prod.inject rx set-ConsD set-append ts xrx*)
qed
} **note** *lin-vars* = *this*
show $(\to_m)^{**}$ (*mp-lr mp*) (*mp-lr* (*match-var-impl mp*)) **unfolding** *mp mp′*(*1*) **using** *lin-vars*
proof (*induct deleted*)
case (*add pair deleted*)
obtain *t xt* **where** *pair*: *pair* = (*t*,*xt*) **by** *force*
hence (*t*,*xt*) ∈# *add-mset pair deleted* **by** *auto*
from *add*(*2*)[*OF this*] *pair*
obtain *x* **where** *add-mset pair deleted* + *mp′* = *add-mset* (*t*, *Var x*) (*deleted* + *mp′*)
and *x*: *x* ∉ ⋃ (*vars* ‘ *snd* ‘ (*mp-mset* (*deleted* + *mp′*)))
and *pair*: *pair* = (*t*, *Var x*)

```
      by auto
    from match-match[OF this(2), of t, folded this(1)]
    have one: add-mset pair deleted + mp′ →m (deleted + mp′) .
    have two: (→m)** (deleted + mp′) mp′
    proof (rule add(1), goal-cases)
      case (1 s yt)
      hence (s,yt) ∈# add-mset pair deleted by auto
      from add(2)[OF this]
      obtain y mp′′ where yt: yt = Var y add-mset pair deleted = add-mset (s,
Var y) mp′′
        y ∉ ⋃ (vars ‘ snd ‘ (mp-mset mp′′ ∪ mp-mset mp′))
        by auto
      from 1[unfolded yt] have y ∈ ⋃ (vars ‘ snd ‘ (mp-mset (deleted + mp′)))
by force
      with x have x ≠ y by auto
      with pair yt have pair ≠ (s,Var y) by auto
      with yt(2) have del: deleted = add-mset (s, Var y) (mp′′ − {#pair#})
        by (meson add-eq-conv-diff)
      show ?case
        by (intro exI conjI, rule yt, rule del, rule contra-subsetD[OF - yt(3)])
         (intro UN-mono, auto dest: in-diffD)
    qed
    from one two show ?case by auto
  qed auto
qed

lemma match-steps-impl: assumes match-steps-impl mp = res
  shows res = Some mp′ ⟹ (→m)** (mp-list mp) (mp-lr mp′) ∧ wf-lr2 mp′
    and res = None ⟹ ∃ mp′. (→m)** (mp-list mp) mp′ ∧ match-fail mp′
proof (atomize (full), goal-cases)
  case 1
  obtain res′ where decomp: decomp-impl mp = res′ by auto
  note res = assms[unfolded match-steps-impl-def decomp]
  note decomp = decomp-impl[OF decomp, of - {#}, unfolded empty-neutral]
  show ?case
  proof (cases res′)
    case None
    with decomp res show ?thesis by auto
  next
    case (Some mp′′)
    with decomp[of mp′′]
    have steps: (→m)** (mp-list mp) (mp-lr mp′′) and wf: wf-lr mp′′ by auto
    from res[unfolded Some] have res: res = Some (match-var-impl mp′′) by auto
    from match-var-impl[OF wf] steps res show ?thesis by auto
  qed
qed

lemma pat-inner-impl: assumes pat-inner-impl p pd = res
  and wf-pat-lr pd
```

and *tvars-pp* (*pat-mset* (*pat-mset-list p* + *pat-lr pd*)) $\subseteq$ *V*
shows *res* = *None* $\Longrightarrow$ (*add-mset* (*pat-mset-list p* + *pat-lr pd*) *P*, *P*) $\in \Rrightarrow^+$
and *res* = *Some p'* $\Longrightarrow$ (*add-mset* (*pat-mset-list p* + *pat-lr pd*) *P*, *add-mset* (*pat-lr p'*) *P*) $\in \Rrightarrow^*$
$\wedge$ *wf-pat-lr p'* $\wedge$ *tvars-pp* (*pat-mset* (*pat-lr p'*)) $\subseteq$ *V*
proof (*atomize*(*full*), *insert assms*, *induct p arbitrary*: *pd res p'*)
case *Nil*
then show *?case* **by** (*auto simp*: *wf-pat-lr-def pat-mset-list-def pat-lr-def*)
next
case (*Cons mp p pd res p'*)
let *?p* = *pat-mset-list p* + *pat-lr pd*
have *id*: *pat-mset-list* (*mp* # *p*) + *pat-lr pd* = *add-mset* (*mp-list mp*) *?p* **unfolding** *pat-mset-list-def* **by** *auto*
show *?case*
proof (*cases match-steps-impl mp*)
case (*Some mp'*)
from *match-steps-impl*(*1*)[*OF Some refl*]
have *steps*: $(\rightarrow_m)^{**}$ (*mp-list mp*) (*mp-lr mp'*) **and** *wf*: *wf-lr2 mp'* **by** *auto*
have *id2*: *pat-mset-list p* + *pat-lr* (*mp'* # *pd*) = *add-mset* (*mp-lr mp'*) *?p* **unfolding** *pat-lr-def* **by** *auto*
from *mp-step-mset-steps-vars*[*OF steps*] *Cons*(*4*)
have *vars*: *tvars-pp* (*pat-mset* (*pat-mset-list p* + *pat-lr* (*mp'* # *pd*))) $\subseteq$ *V*
unfolding *id2* **by** (*auto simp*: *tvars-pp-def pat-mset-list-def*)
note *steps* = *mp-step-mset-cong*[*OF steps*, *of ?p P*, *folded id*]
note *res* = *Cons*(*2*)[*unfolded pat-inner-impl.simps Some option.simps*]
show *?thesis*
proof (*cases empty-lr mp'*)
case *False*
with *Cons*(*3*) *wf* **have** *wf*: *wf-pat-lr* (*mp'* # *pd*) **unfolding** *wf-pat-lr-def* **by** *auto*
from *res False* **have** *pat-inner-impl p* (*mp'* # *pd*) = *res* **by** *auto*
from *Cons*(*1*)[*OF this wf*, *of p'*, *OF vars*, *unfolded id2*] *steps*
show *?thesis* **by** *auto*
next
case *True*
with *wf* **have** *id3*: *mp-lr mp'* = {#} **unfolding** *wf-lr2-def empty-lr-def* **by** (*cases mp'*, *auto simp*: *mp-lr-def mp-rx-def List.maps-def*)
from *True res* **have** *res*: *res* = *None* **by** *auto*
have (*add-mset* (*add-mset* (*mp-lr mp'*) *?p*) *P*, *P*) $\in$ *P-step*
unfolding *id3 P-step-def* **using** *P-simp-pp*[*OF pat-remove-pp*[*of ?p*], *of P*] **by** *auto*
with *res steps* **show** *?thesis* **by** *auto*
qed
next
case *None*
from *match-steps-impl*(*2*)[*OF None refl*] **obtain** *mp'* **where**
$(\rightarrow_m)^{**}$ (*mp-list mp*) *mp'* **and** *fail*: *match-fail mp'* **by** *auto*
note *steps* = *mp-step-mset-cong*[*OF this*(*1*), *of ?p P*, *folded id*]
from *P-simp-pp*[*OF pat-remove-mp*[*OF fail*, *of ?p*], *of P*]

have (*add-mset* (*add-mset mp′ ?p*) *P*, *add-mset ?p P*) ∈ *P-step*
unfolding *P-step-def* **by** *auto*
with *steps* **have** *steps*: (*add-mset* (*pat-mset-list* (*mp* # *p*) + *pat-lr pd*) *P*, *add-mset ?p P*) ∈ *P-step*^* **by** *auto*
note *res* = *Cons*(*2*)[*unfolded pat-inner-impl.simps None option.simps*]
have *vars*: *tvars-pp* (*pat-mset* (*pat-mset-list p* + *pat-lr pd*)) ⊆ *V*
using *Cons*(*4*) **unfolding** *tvars-pp-def pat-mset-list-def* **by** *auto*
from *Cons*(*1*)[*OF res Cons*(*3*), *of p′*, *OF vars*] *steps*
show *?thesis* **by** *auto*
qed
qed

lemma *pat-mset-list*: *pat-mset* (*pat-mset-list p*) = *pat-list p*
unfolding *pat-list-def pat-mset-list-def* **by** (*auto simp*: *image-comp*)

Main simulation lemma for a single *pat-impl* step.

lemma *pat-impl*: **assumes** *pat-impl n p* = *res*
and *vars*: *fst* ‘ *tvars-pp* (*pat-list p*) ⊆ {..<*n*}
shows *res* = *None* ⟹ ∃ *p′*. (*add-mset* (*pat-mset-list p*) *P*, *add-mset p′ P*) ∈ ⇛* ∧ *pat-fail p′*
and *res* = *Some ps* ⟹ (*add-mset* (*pat-mset-list p*) *P*, *mset* (*map pat-mset-list ps*) + *P*) ∈ ⇛+
∧ *fst* ‘ *tvars-pp* (⋃ (*pat-list* ‘ *set ps*)) ⊆ {..<*n*+*m*}
proof (*atomize*(*full*), *goal-cases*)
case *1*
have *wf*: *wf-pat-lr* [] **unfolding** *wf-pat-lr-def* **by** *auto*
have *fst* ‘ *tvars-pp* (*pat-mset* (*pat-mset-list p*)) ⊆ {..<*n*}
using *vars* **unfolding** *pat-mset-list* **.**
hence *vars*: *tvars-pp* (*pat-mset* (*pat-mset-list p*)) ⊆ {..<*n*} × *UNIV* **by** *force*
have *pat-mset-list p* + *pat-lr* [] = *pat-mset-list p* **unfolding** *pat-lr-def* **by** *auto*
note *pat-inner* = *pat-inner-impl*[*OF refl wf*, *of p*, *unfolded this*, *OF vars*]
note *res* = *assms*(*1*)[*unfolded pat-impl-def*]
show *?case*
proof (*cases pat-inner-impl p* [])
case *None*
from *pat-inner*(*1*)[*OF this*, *of P*] *res*[*unfolded None option.simps*] *vars*
show *?thesis* **by** (*auto simp*: *tvars-pp-def*)
next
case (*Some p′*)
from *pat-inner*(*2*)[*OF this*, *of P*]
have *steps*: (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*pat-lr p′*) *P*) ∈ ⇛* **and** *wf*: *wf-pat-lr p′*
and *varsp′*: *tvars-pp* (*pat-mset* (*pat-lr p′*)) ⊆ {..<*n*} × *UNIV* **by** *auto*
note *res* = *res*[*unfolded Some option.simps*]
show *?thesis*
proof (*cases* ∀ *mp*∈*set p′*. *snd* (*snd mp*))
case *True*
with *res* **have** *res*: *res* = *None* **by** *auto*
have *pat-fail* (*pat-lr p′*)

proof (*intro pat-failure′ ballI*)
fix *mps*
assume *mps* ∈ *pat-mset* (*pat-lr p′*)
then obtain *mp* **where** *mem*: *mp* ∈ *set p′* **and** *mps*: *mps* = *mp-mset* (*mp-lr mp*) **by** (*auto simp*: *pat-lr-def*)
obtain *lx rx b* **where** *mp*: *mp* = (*lx*,*rx*,*b*) **by** (*cases mp*, *auto*)
from *mp mem True* **have** *b* **by** *auto*
with *wf*[*unfolded wf-pat-lr-def*, *rule-format*, *OF mem*, *unfolded wf-lr2-def mp split*]
have *inf-var-conflict* (*set-mset* (*mp-rx* (*rx*,*b*))) **unfolding** *wf-rx-def wf-rx2-def* **by** (*auto split*: *if-splits*)
thus *inf-var-conflict mps* **unfolding** *mps mp-lr-def mp split*
unfolding *inf-var-conflict-def* **by** *fastforce*
qed
with *steps res*
show *?thesis* **by** *auto*
next
case *False*
define *p′l* **where** *p′l* = *map mp-lr-list p′*
define *x* **where** *x* = *find-var p′*
define *ps* **where** *ps* = *map* (*λτ*. *subst-pat-problem-list τ p′l*) (*τs-list n x*)
have *id*: *pat-lr p′* = *pat-mset-list p′l* **unfolding** *pat-mset-list-def pat-lr-def p′l-def map-map o-def*
by (*intro arg-cong*[*of - - mset*] *map-cong refl*, *auto simp*: *mp-lr-def mp-lr-list-def mp-rx-def mp-rx-list-def*)
from *False* **have** (∀ *mp*∈*set p′*. *snd* (*snd mp*)) = *False* **by** *auto*
from *res*[*unfolded this if-False Let-def*, *folded p′l-def x-def*, *folded ps-def*]
have *res*: *res* = *Some ps* **by** *auto*
have *step*: (*add-mset* (*pat-lr p′*) *P*, *mset* (*map pat-mset-list ps*) + *P*) ∈ ⇛
unfolding *P-step-def*
proof (*standard*, *unfold split*, *intro P-simp-pp*)
note *x* = *x-def*[*unfolded find-var-def*]
let *?concat* = *concat* (*map* (*λ* (*lx*,-). *lx*) *p′*)
have *disj*: *tvars-disj-pp* {*n*..<*n* + *m*} (*pat-mset* (*pat-lr p′*))
using *varsp′* **unfolding** *tvars-pp-def tvars-disj-pp-def tvars-mp-def* **by** *force*
have *subst*: *map* (*λτ*. *subst-pat-problem-mset τ* (*pat-lr p′*)) (*τs-list n x*) = *map pat-mset-list ps*
unfolding *id*
unfolding *ps-def subst-pat-problem-list-def subst-pat-problem-mset-def subst-match-problem-mset-def*
subst-match-problem-list-def map-map o-def
by (*intro list.map-cong0*, *auto simp*: *pat-mset-list-def o-def image-mset.compositionality*)
show *pat-lr p′* $\Rightarrow_m$ *mset* (*map pat-mset-list ps*)
proof (*cases ?concat*)
case (*Cons pair list*)
with *x* **obtain** *t* **where** *concat*: *?concat* = (*x*,*t*) # *list* **by** (*cases pair*, *auto*)
hence (*x*,*t*) ∈ *set ?concat* **by** *auto*

then obtain *mp* **where** $mp \in set\ p'$ **and** $(x,t) \in set\ ((\lambda\ (lx,\text{-}).\ lx)\ mp)$ **by** *auto*

then obtain *lx rx* **where** *mem*: $(lx,rx) \in set\ p'$ **and** *xt*: $(x,t) \in set\ lx$ **by** *auto*

from *wf mem* **have** *wf*: *wf-lx lx* **unfolding** *wf-pat-lr-def wf-lr2-def* **by** *auto*

with *xt* **have** *t*: *is-Fun t* **unfolding** *wf-lx-def* **by** *auto*

from *mem* **obtain** p'' **where** *pat*: $pat\text{-}lr\ p' = add\text{-}mset\ (mp\text{-}lr\ (lx,rx))\ p''$

unfolding *pat-lr-def* **by** *simp* (*metis in-map-mset mset-add set-mset-mset*)

from *xt* **have** *xt*: $(Var\ x,\ t) \in\#\ mp\text{-}lr\ (lx,rx)$ **unfolding** *mp-lr-def* **by** *force*

from *pat-instantiate*[*OF - disjI1*[*OF conjI*[*OF xt t*]], *of n* p'', *folded pat*, *OF disj*]

show *?thesis* **unfolding** *subst* **.**

next

case *Nil*

let $?fp = filter\ (Not \circ snd \circ snd)\ p'$

from *False* **have** $set\ ?fp \neq \{\}$ **unfolding** *o-def filter-empty-conv set-empty*

by *auto*

then obtain *mp* p'' **where** *fp*: $?fp = mp\ \#\ p''$ **by** (*cases ?fp*) *auto*

obtain *lx rx b* **where** *mp*: $mp = (lx,rx,b)$ **by** (*cases mp*) *auto*

have *mpp*: $mp \in set\ p'$ **using** *arg-cong*[*OF fp*, *of set*] **by** *auto*

from *mp mpp Nil* **have** *lx*: $lx = []$ **by** *auto*

from *fp* **have** $(lx,rx,b) \in set\ ?fp$ **unfolding** *mp* **by** *auto*

hence $\neg\ b$ **unfolding** *o-def* **by** *auto*

with *mp lx* **have** *mp*: $mp = ([],rx,False)$ **by** *auto*

from *wf mpp* **have** *wf*: *wf-lr2 mp* **and** *ne*: $\neg$ *empty-lr mp* **unfolding** *wf-pat-lr-def* **by** *auto*

from *wf*[*unfolded wf-lr2-def mp split*] *mp*

have *wf*: *wf-rx2* (*rx*, *False*) **and** *mp*: $mp = ([],rx,False)$ **by** *auto*

from *ne*[*unfolded empty-lr-def mp split*] **obtain** *y ts* rx'

where *rx*: $rx = (y,ts)\ \#\ rx'$ **by** (*cases rx*, *auto*)

from *wf*[*unfolded wf-rx2-def*] **have** *ninf*: $\neg$ *inf-var-conflict* (*mp-mset* (*mp-rx* (*rx*, *False*)))

and *wf*: *wf-ts2 ts* **unfolding** *rx* **by** *auto*

from *wf*[*unfolded wf-ts2-def*] **obtain** *s t* ts' **where** *ts*: $ts = s\ \#\ t\ \#\ ts'$ **and**

diff: $s \neq t$ **and** *conf*: $conflicts\ s\ t \neq None$

by (*cases ts*; *cases tl ts*, *auto*)

from *conf* **obtain** *xs* **where** *conf*: $conflicts\ s\ t = Some\ xs$ **by** (*cases conflicts s t*, *auto*)

with *conflicts*(5)[*of s t*] *diff* **have** $xs \neq []$ **by** *auto*

with *x*[*unfolded Nil list.simps fp list.sel mp split Let-def rx ts conf option.sel*]

obtain xs' **where** *xs*: $xs = x\ \#\ xs'$ **by** (*cases xs*) *auto*

from *conf xs* **have** *confl*: *Conflict-Var s t x* **by** *auto*

from *ts rx* **have** *sty*: $(s,\ Var\ y) \in\#\ mp\text{-}rx\ (rx,\ False)\ (t,\ Var\ y) \in\#\ mp\text{-}rx\ (rx,False)$

by (*auto simp*: *mp-rx-def List.maps-def*)

with *confl ninf* **have** $\neg$ *inf-sort* (*snd x*) **unfolding** *inf-var-conflict-def* **by** *blast*

with *sty confl rx* **have** *main*: $(s,\ Var\ y) \in\#$ *mp-lr mp* $\wedge$ $(t,\ Var\ y) \in\#$ *mp-lr mp* $\wedge$ *Conflict-Var s t x* $\wedge$ $\neg$ *inf-sort* (*snd x*)
unfolding *mp* **by** (*auto simp*: *mp-lr-def*)
from *mpp* **obtain** p'' **where** *pat*: *pat-lr* p' = *add-mset* (*mp-lr mp*) p''
unfolding *pat-lr-def* **by** *simp* (*metis in-map-mset mset-add set-mset-mset*)
from *pat-instantiate*[*OF - disjI2*[*OF main*], *of n* p'', *folded pat*, *OF disj*]
show *?thesis* **unfolding** *subst* **.**
qed
qed
have *fst* ‘ *tvars-pp* ($\bigcup$ (*pat-list* ‘ *set ps*)) $\subseteq$ {..<*n* + *m*}
proof
fix *yn*
assume *yn* $\in$ *fst* ‘ *tvars-pp* ($\bigcup$ (*pat-list* ‘ *set ps*))
then obtain *pi y mp* **where** *pi*: *pi* $\in$ *set ps* **and** *mp*: *mp* $\in$ *set pi* **and** *y*: *y* $\in$ *tvars-mp* (*set mp*) **and** *yn*: *yn* = *fst y*
unfolding *tvars-pp-def pat-list-def* **by** *force*
from *pi*[*unfolded ps-def set-map subst-pat-problem-list-def subst-match-problem-list-def*, *simplified*]
obtain τ **where** *tau*: $\tau \in$ *set* (τ*s-list n x*) **and** *pi*: *pi* = *map* (*map* (*subst-left* τ)) $p'l$ **by** *auto*
from *tau*[*unfolded* τ*s-list-def*]
obtain *info* **where** *info* $\in$ *set* (*Cl* (*snd x*)) **and** *tau*: τ = τ*c n x info* **by** *auto*
from *Cl-len*[*of snd x*] *this*(*1*) **have** *len*: *length* (*snd info*) $\leq$ *m* **by** *force*
from *mp*[*unfolded pi set-map*] **obtain** mp' **where** mp': $mp' \in$ *set* $p'l$ **and** *mp*: *mp* = *map* (*subst-left* τ) mp' **by** *auto*
from *y*[*unfolded mp tvars-mp-def image-comp o-def set-map*]
obtain *pair* **where** $*$: *pair* $\in$ *set* mp' *y* $\in$ *vars* (*fst* (*subst-left* τ *pair*)) **by** *auto*
obtain *s t* **where** *pair*: *pair* = (*s*,*t*) **by** *force*
from $*$[*unfolded pair*] **have** *st*: (*s*,*t*) $\in$ *set* mp' **and** *y*: *y* $\in$ *vars* ($s \cdot \tau$) **unfolding** *subst-left-def* **by** *auto*
from *y*[*unfolded vars-term-subst*, *simplified*] **obtain** *z* **where** *z*: *z* $\in$ *vars s* **and** *y*: *y* $\in$ *vars* (τ *z*) **by** *auto*
obtain *f ss* **where** *info*: *info* = (*f*,*ss*) **by** (*cases info*, *auto*)
with *len* **have** *len*: *length ss* $\leq$ *m* **by** *auto*
define *ts* :: ($'f$,-)*term list* **where** *ts* = *map Var* (*zip* [*n*..<*n* + *length ss*] *ss*)
from *tau*[*unfolded* τ*c-def info split*] **have** *tau*: τ = *subst x* (*Fun f ts*) **unfolding** *ts-def* **by** *auto*
have *fst* ‘ *vars* (*Fun f ts*) $\subseteq$ {..< *n* + *length ss*} **unfolding** *ts-def* **by** (*auto simp*: *set-zip*)
also have ... $\subseteq$ {..< *n* + *m*} **using** *len* **by** *auto*
finally have *subst*: *fst* ‘ *vars* (*Fun f ts*) $\subseteq$ {..< *n* + *m*} **by** *auto*
show *yn* $\in$ {..<*n* + *m*}
proof (*cases z* = *x*)
case *True*
with *y subst tau yn* **show** *?thesis* **by** *auto*
next
case *False*

hence τ z = *Var z* **unfolding** *tau* **by** (*auto simp*: *subst-def*)
with *y* **have** $z = y$ **by** *auto*
with *z* **have** *y*: $y \in$ *vars s* **by** *auto*
with *st* **have** $y \in$ *tvars-mp* (*set mp'*) **unfolding** *tvars-mp-def* **by** *force*
with *mp'* **have** $y \in$ *tvars-pp* (*set ‘ set p'l*) **unfolding** *tvars-pp-def* **by** *auto*
also have ... = *tvars-pp* (*pat-mset* (*pat-mset-list p'l*))
by (*rule arg-cong*[*of - - tvars-pp*], *auto simp*: *pat-mset-list-def image-comp*)
also have ... = *tvars-pp* (*pat-mset* (*pat-lr p'*)) **unfolding** *id*[*symmetric*]
by *simp*
also have ... $\subseteq$ {..<*n*} $\times$ *UNIV* **using** *varsp'* **.**
finally show *?thesis* **using** *yn* **by** *auto*
qed
qed
with *step steps res* **show** *?thesis* **by** *auto*
qed
qed
qed

The simulation property for *pats-impl*, proven by induction on the terminating relation of the multiset-implementation.

lemma *pats-impl-P-step*: **assumes** *Ball* (*set ps*) (λ *p*. *fst ‘ tvars-pp* (*pat-list p*) $\subseteq$ {..<*n*})
shows
— if result is True, then one can reach empty set
pats-impl n ps $\Longrightarrow$ (*pats-mset-list ps*, {#}) $\in$ $\Rrightarrow^*$
— if result is False, then one can reach bottom
$\neg$ *pats-impl n ps* $\Longrightarrow$ (*pats-mset-list ps*, *bottom-mset*) $\in$ $\Rrightarrow^*$
proof (*atomize*(*full*), *insert assms*, *induct ps arbitrary*: *n rule*: *SN-induct*[*OF SN-inv-image*[*OF SN-imp-SN-trancl*[*OF SN-P-step*]], *of pats-mset-list*])
case (*1 ps n*)
show *?case*
proof (*cases ps*)
case *Nil*
show *?thesis* **unfolding** *pats-impl.simps*[*of n ps*] **unfolding** *Nil* **by** *auto*
next
case (*Cons p ps1*)
hence *id*: *pats-mset-list ps* = *add-mset* (*pat-mset-list p*) (*pats-mset-list ps1*) **by** *auto*
note *res* = *pats-impl.simps*[*of n ps*, *unfolded Cons list.simps*, *folded Cons*]
from *1*(*2*)[*rule-format*, *of p*] *Cons* **have** *fst ‘ tvars-pp* (*pat-list p*) $\subseteq$ {..<*n*} **by** *auto*
note *pat-impl* = *pat-impl*[*OF refl this*]
show *?thesis*
proof (*cases pat-impl n p*)
case *None*
with *res* **have** *res*: *pats-impl n ps* = *False* **by** *auto*
from *pat-impl*(*1*)[*OF None*, *of pats-mset-list ps1*, *folded id*]
obtain *p'* **where** *steps*: (*pats-mset-list ps*, *add-mset p'* (*pats-mset-list ps1*)) $\in$ $\Rrightarrow^*$ **and** *fail*: *pat-fail p'*

by *auto*
show *?thesis*
proof (*cases add-mset* p' (*pats-mset-list ps1*) = *bottom-mset*)
case *True*
with *res steps* **show** *?thesis* **by** *auto*
next
case *False*
from *P-failure*[*OF fail False*]
have (*add-mset* p' (*pats-mset-list ps1*), *bottom-mset*) $\in \Rrightarrow$ **unfolding** *P-step-def* **by** *auto*
with *steps res* **show** *?thesis* **by** *simp*
qed
next
case (*Some ps2*)
with *res* **have** *res*: *pats-impl n ps* = *pats-impl* $(n + m)$ (*ps2* @ *ps1*) **by** *auto*
from *pat-impl*(*2*)[*OF Some*, *of pats-mset-list ps1*, *folded id*]
have *steps*: (*pats-mset-list ps*, *mset* (*map pat-mset-list* (*ps2* @ *ps1*))) $\in \Rrightarrow^+$
and *vars*: *fst* ‘ *tvars-pp* ($\bigcup$ (*pat-list* ‘ *set ps2*)) $\subseteq$ {..<$n + m$} **by** *auto*
hence *rel*: (*ps*, *ps2* @ *ps1*) $\in$ *inv-image* (*P-step*$^+$) *pats-mset-list* **by** *auto*
have *vars*: $\forall$ *p*$\in$*set* (*ps2* @ *ps1*). *fst* ‘ *tvars-pp* (*pat-list p*) $\subseteq$ {..<$n + m$}
proof
fix *p*
assume $p \in$ *set* (*ps2* @ *ps1*)
hence $p \in$ *set ps2* $\vee$ $p \in$ *set ps1* **by** *auto*
thus *fst* ‘ *tvars-pp* (*pat-list p*) $\subseteq$ {..<$n + m$}
proof
assume $p \in$ *set ps2*
hence *fst* ‘ *tvars-pp* (*pat-list p*) $\subseteq$ *fst* ‘ *tvars-pp* ($\bigcup$ (*pat-list* ‘ *set ps2*))
unfolding *tvars-pp-def* **by** *auto*
with *vars* **show** *?thesis* **by** *auto*
next
assume $p \in$ *set ps1*
hence $p \in$ *set ps* **unfolding** *Cons* **by** *auto*
from *1*(*2*)[*rule-format*, *OF this*] **show** *?thesis* **by** *auto*
qed
qed
show *?thesis* **using** *1*(*1*)[*OF rel vars*] *steps* **unfolding** *res*[*symmetric*] **by** *auto*
qed
qed
qed

Consequence: partial correctness of the list-based implementation on well-formed inputs

theorem *pats-impl*: **assumes** *wf*: $\forall$ *pp* $\in$ *pat-list* ‘ *set P*. *wf-pat pp*
and *n*: $\forall$ *p*$\in$*set P*. *fst* ‘ *tvars-pp* (*pat-list p*) $\subseteq$ {..<n}
shows *pats-impl n P* $\longleftrightarrow$ *pats-complete* (*pat-list* ‘ *set P*)
proof −
from *wf* **have** *wf*: *wf-pats* (*pat-list* ‘ *set P*) **by** (*auto simp*: *wf-pats-def*)

have *id*: *pats-mset* (*pats-mset-list P*) = *pat-list* ‘ *set P* **unfolding** *pat-list-def*
by (*auto simp*: *pat-mset-list-def image-comp*)
{
assume *pats-impl n P*
from *pats-impl-P-step*(*1*)[*OF n this*]
have (*pats-mset-list P*, {#}) $\in \Rrightarrow^*$ **by** *auto*
from *P-steps-pcorrect*[*OF - this*, *unfolded id*, *OF wf*]
have *pats-complete* (*pat-list* ‘ *set P*) **by** *auto*
}
moreover
{
assume ¬ *pats-impl n P*
from *pats-impl-P-step*(*2*)[*OF n this*]
have (*pats-mset-list P*, {#{#}#}) $\in \Rrightarrow^*$ **by** *auto*
from *P-steps-pcorrect*[*OF - this*, *unfolded id*, *OF wf*]
have ¬ *pats-complete* (*pat-list* ‘ *set P*) **by** *auto*
}
ultimately show *?thesis* **by** *auto*
qed

corollary *pat-complete-impl*:
assumes *wf*: *snd* ‘ $\bigcup$ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat P*))) $\subseteq$ *S*
shows *pat-complete-impl P* $\longleftrightarrow$ *pats-complete* (*pat-list* ‘ *set P*)
proof −
have *wf*: *Ball* (*pat-list* ‘ *set P*) *wf-pat*
unfolding *pat-list-def wf-pat-def wf-match-def tvars-mp-def* **using** *wf*[*unfolded set-concat image-comp*] **by** *force*
let *?l* = (*List.maps* (*map fst o vars-term-list o fst*) (*concat* (*concat P*)))
define *n* **where** *n* = *Suc* (*max-list ?l*)
have *n*: $\forall$ *p*$\in$*set P*. *fst* ‘ *tvars-pp* (*pat-list p*) $\subseteq$ {..<*n*}
proof (*intro ballI subsetI*)
fix *p x*
assume *p* $\in$ *set P* **and** *x* $\in$ *fst* ‘ *tvars-pp* (*pat-list p*)
hence *x* $\in$ *set ?l* **unfolding** *List.maps-def tvars-pp-def tvars-mp-def pat-list-def*
by *force*
from *max-list*[*OF this*] **have** *x* < *n* **unfolding** *n-def* **by** *auto*
thus *x* $\in$ {..<*n*} **by** *auto*
qed
have *pat-complete-impl P* = *pats-impl n P*
unfolding *pat-complete-impl-def n-def Let-def* **..**
from *pats-impl*[*OF wf n*, *folded this*]
show *?thesis* **.**
qed
end

6.3 Getting the result outside the locale with assumptions

We next lift the results for the list-based implementation out of the locale. Here, we use the existing algorithms to decide non-empty sorts *decide-nonempty-sorts* and to compute the infinite sorts *compute-inf-sorts*.

context *pattern-completeness-context*
begin
lemma *pat-complete-impl-wrapper*: **assumes** *C-Cs*: $C = \mathit{map\text{-}of}\ Cs$
  and *dist*: $\mathit{distinct}\ (\mathit{map}\ \mathit{fst}\ Cs)$
  and *inhabited*: $\mathit{decide\text{-}nonempty\text{-}sorts}\ Sl\ Cs = \mathit{None}$
  and *S-Sl*: $S = \mathit{set}\ Sl$
  and *inf-sort*: $\mathit{inf\text{-}sort} = (\lambda\ s.\ s \in \mathit{compute\text{-}inf\text{-}sorts}\ Cs)$
  and *C*: $\bigwedge f\ \sigma s\ \sigma.\ ((f,\sigma s),\sigma) \in \mathit{set}\ Cs \Longrightarrow \mathit{length}\ \sigma s \leq m \wedge \mathit{set}\ (\sigma\ \#\ \sigma s) \subseteq S$
  and *Cl*: $\bigwedge s.\ Cl\ s = \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ ((=)\ s\ o\ \mathit{snd})\ Cs)$
  and *P*: $\mathit{snd}\ `\ \bigcup\ (\mathit{vars}\ `\ \mathit{fst}\ `\ \mathit{set}\ (\mathit{concat}\ (\mathit{concat}\ P))) \subseteq S$
  shows $\mathit{pat\text{-}complete\text{-}impl}\ P = \mathit{pats\text{-}complete}\ (\mathit{pat\text{-}list}\ `\ \mathit{set}\ P)$
proof −
  from *decide-nonempty-sorts*(*1*)[*OF dist C-Cs*[*symmetric*] *inhabited*, *folded S-Sl*]
  have *S*: $\bigwedge \sigma.\ \sigma \in S \Longrightarrow \exists t.\ t : \sigma\ \mathit{in}\ \mathcal{T}(C,\mathit{EMPTY})$
    $\bigwedge \sigma.\ \sigma \in S \Longrightarrow \exists t.\ t : \sigma\ \mathit{in}\ \mathcal{T}(C,\mathit{EMPTYn})$ **unfolding** *EMPTY-def EMPTYn-def* **by** *auto*
  {
    fix *f ss s*
    assume $f : ss \to s\ \mathit{in}\ C$
    hence $((f,ss),s) \in \mathit{set}\ Cs$ **unfolding** *C-Cs* **by** (*auto dest*!: *hastype-in-ssigD map-of-SomeD*)
    from *C*[*OF this*] **have** $\mathit{insert}\ s\ (\mathit{set}\ ss) \subseteq S\ \mathit{length}\ ss \leq m$ **by** *auto*
  } **note** *Cons* = *this*
  {
    fix *f ss s*
    assume $(f,ss) \in \mathit{set}\ (Cl\ s)$
    hence $((f,ss),s) \in \mathit{set}\ Cs$ **unfolding** *Cl* **by** *auto*
    from *C*[*OF this*] **have** $\mathit{length}\ ss \leq m$ **by** *auto*
  }
  hence *m*: $\forall a{\in}\mathit{length}\ `\ \mathit{snd}\ `\ \mathit{set}\ (Cl\ s).\ a \leq m$ **for** *s* **by** *auto*
  have *En*: $\mathit{EMPTYn} = \emptyset$ **unfolding** *EMPTYn-def* **by** *auto*
  have $\forall f\ ss\ s\ s'.\ f : ss \to s\ \mathit{in}\ C \longrightarrow s' \in \mathit{set}\ ss \longrightarrow (\exists t.\ t : s'\ \mathit{in}\ \mathcal{T}(C,\mathit{EMPTYn}))$

  proof (*intro allI impI*)
    fix *f ss s s′*
    assume $f : ss \to s\ \mathit{in}\ C$ **and** $s' \in \mathit{set}\ ss$
    hence $s' \in S$ **using** *Cons*(*1*)[*of f ss s*] **by** (*auto simp*: *hastype-in-ssig-def*)
    from *S*[*OF this*] **show** $\exists t.\ t : s'\ \mathit{in}\ \mathcal{T}(C,\mathit{EMPTYn})$ **by** *auto*
  qed
  from *compute-inf-sorts*[*OF En C-Cs this dist*] *inf-sort*
  have *inf-sort*: $\mathit{inf\text{-}sort}\ s = (\neg\ \mathit{bdd\text{-}above}\ (\mathit{size}\ `\ \{t.\ t : s\ \mathit{in}\ \mathcal{T}(C,\mathit{EMPTYn})\}))$ **for** *s* **unfolding** *inf-sort* **by** *auto*
  have *Cl*: $\mathit{set}\ (Cl\ s) = \{(f,ss).\ f : ss \to s\ \mathit{in}\ C\}$ **for** *s*
    unfolding *Cl set-map o-def C-Cs* **using** *dist*

by (*force simp*: *hastype-in-ssig-def*)
interpret *pattern-completeness-context-with-assms*
apply *unfold-locales*
subgoal by (*rule S*(*1*))
subgoal by (*rule Cons*)
subgoal by (*rule Cons*)
subgoal by (*rule inf-sort*)
subgoal by (*rule Cl*)
subgoal by (*rule m*)
done
show *?thesis* **by** (*rule pat-complete-impl*[*OF P*])
qed
end

Next we are also leaving the locale that fixed the common parameters, and chooses suitable values.

extract all sorts from a ssignature (input and target sorts)

definition *sorts-of-ssig-list* :: $(('f \times 's\ list) \times 's) list \Rightarrow 's\ list$ **where**
sorts-of-ssig-list Cs = *remdups* (*List.maps* (λ ((*f*,*ss*),*s*). *s* # *ss*) *Cs*)

definition *decide-pat-complete* :: $(('f \times 's\ list) \times 's) list \Rightarrow ('f,'v,'s) pats\text{-}problem\text{-}list \Rightarrow bool$ **where**
decide-pat-complete Cs P = (*let Sl* = *sorts-of-ssig-list Cs*;
m = *max-list* (*map* (*length o snd o fst*) *Cs*);
Cl = (λ *s*. *map fst* (*filter* ((=) *s* $\circ$ *snd*) *Cs*));
IS = *compute-inf-sorts Cs*
in pattern-completeness-context.pat-complete-impl m Cl (λ *s*. *s* $\in$ *IS*)) *P*

abbreviation (*input*) *pat-complete* **where**
pat-complete $\equiv$ *pattern-completeness-context.pat-complete*

abbreviation (*input*) *pats-complete* **where**
pats-complete $\equiv$ *pattern-completeness-context.pats-complete*

Finally: a pattern completeness decision procedure for arbitrary inputs, assuming sensible inputs

theorem *decide-pat-complete*: **assumes** *C-Cs*: *C* = *map-of Cs*
and *dist*: *distinct* (*map fst Cs*)
and *non-empty-sorts*: *decide-nonempty-sorts* (*sorts-of-ssig-list Cs*) *Cs* = *None*
and *S*: *S* = *set* (*sorts-of-ssig-list Cs*)
and *P*: *snd* ‘ $\bigcup$ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat P*))) $\subseteq$ *S*
shows *decide-pat-complete Cs P* = *pats-complete S C* (*pat-list* ‘ *set P*)
unfolding *decide-pat-complete-def Let-def*
proof (*rule pattern-completeness-context.pat-complete-impl-wrapper*[*OF C-Cs dist non-empty-sorts S refl - refl P*])
fix *f* σs σ
assume *mem*: ((*f*, σs), σ) $\in$ *set Cs*
hence *length* σs $\in$ *set* (*map* (*length* $\circ$ *snd* $\circ$ *fst*) *Cs*) **by** *force*

from *max-list*[*OF this*] *mem*
show *length* $\sigma s \le$ *max-list* (*map* (*length* $\circ$ *snd* $\circ$ *fst*) *Cs*) $\wedge$ *set* (σ # σs) $\subseteq S$
unfolding *S sorts-of-ssig-list-def List.maps-def* **by** *force*
qed

end

7 Pattern-Completeness and Related Properties

We use the core decision procedure for pattern completeness and connect it to other properties like pattern completeness of programs (where the lhss are given), or (strong) quasi-reducibility.

theory *Pattern-Completeness*
imports
Pattern-Completeness-List
Show.Shows-Literal
Certification-Monads.Check-Monad
begin

A pattern completeness decision procedure for a set of lhss

definition *basic-terms* :: $('f,'s)ssig \Rightarrow ('f,'s)ssig \Rightarrow ('v \rightharpoonup 's) \Rightarrow ('f,'v)term\ set$ ($\mathcal{B}'$(-,-,-')) **where**
$\mathcal{B}(C,D,V) = \{$ *Fun f ts* | *f ss s ts* . *f* : *ss* $\to$ *s in D* $\wedge$ *ts* $:_l$ *ss in* $\mathcal{T}(C,V)\}$

definition *matches* :: $('f,'v)term \Rightarrow ('f,'v)term \Rightarrow bool$ (**infix** *matches 50*) **where**
l matches t = ($\exists$ σ. $t = l \cdot \sigma$)

definition *pat-complete-lhss* :: $('f,'s)ssig \Rightarrow ('f,'s)ssig \Rightarrow ('f,'v)term\ set \Rightarrow bool$ **where**
pat-complete-lhss C D L = ($\forall$ $t \in \mathcal{B}(C,D,\emptyset)$. $\exists\, l \in L$. *l matches t*)

definition *decide-pat-complete-lhss* ::
$(('f \times 's\ list) \times 's)list \Rightarrow (('f \times 's\ list) \times 's)list \Rightarrow ('f,'v)term\ list \Rightarrow showsl + bool$ **where**
decide-pat-complete-lhss C D lhss = *do* {
check (*distinct* (*map fst C*)) (*showsl-lit* (*STR ''constructor information contains duplicate''*));
check (*distinct* (*map fst D*)) (*showsl-lit* (*STR ''defined symbol information contains duplicate''*));
let S = *sorts-of-ssig-list C*;
check-allm (λ ((*f*,*ss*),-). *check-allm* (λ *s*. *check* ($s \in$ *set S*)
(*showsl-lit* (*STR ''a defined symbol has argument sort that is not known in constructors''*))) *ss*) *D*;
(*case* (*decide-nonempty-sorts S C*) *of None* $\Rightarrow$ *return* () | *Some s* $\Rightarrow$ *error* (*showsl-lit* (*STR ''some sort is empty''*)));
let pats = [*Fun f* (*map Var* (*zip* [*0..*<*length ss*] *ss*)). ((*f*,*ss*),*s*) $\leftarrow$ *D*];
let P = [[[(*pat*,*lhs*)]. *lhs* $\leftarrow$ *lhss*]. *pat* $\leftarrow$ *pats*];

return (*decide-pat-complete C P*)
}

theorem *decide-pat-complete-lhss*:
assumes *decide-pat-complete-lhss C D* (*lhss* :: (*'f*,*'v*)*term list*) = *return b*
shows *b* = *pat-complete-lhss* (*map-of C*) (*map-of D*) (*set lhss*)
proof −
let *?EMPTY* = *pattern-completeness-context.EMPTY*
let *?cg-subst* = *pattern-completeness-context.cg-subst*
let *?C* = *map-of C*
let *?D* = *map-of D*
define *S* **where** *S* = *sorts-of-ssig-list C*
define *pats* **where** *pats* = *map* (λ ((*f*,*ss*),*s*). *Fun f* (*map Var* (*zip* [*0*..<*length ss*] *ss*))) *D*
define *P* **where** *P* = *map* (λ *pat*. *map* (λ *lhs*. [(*pat*,*lhs*)]) *lhss*) *pats*
let *?match-lhs* = λ*t*. ∃ *l* ∈ *set lhss*. *l matches t*
note *ass* = *assms*(*1*)[*unfolded decide-pat-complete-lhss-def*, *folded S-def*,
unfolded Let-def, *folded pats-def*, *folded P-def*, *simplified*]
from *ass* **have** *dec*: *decide-nonempty-sorts S C* = *None* (**is** *?e* = -) **by** (*cases ?e*, *auto*)
note *ass* = *ass*[*unfolded dec*, *simplified*]
from *ass* **have** *b*: *b* = *decide-pat-complete C P* **and** *dist*: *distinct* (*map fst C*) *distinct* (*map fst D*) **by** *auto*
have *b* = *decide-pat-complete C P* **by** *fact*
also have ... = *pats-complete* (*set S*) *?C* (*pat-list* ‘ *set P*)
proof (*rule decide-pat-complete*[*OF refl dist*(*1*) *dec*[*unfolded S-def*]], *unfold S-def*[*symmetric*])
{
fix *i si f ss s*
assume *mem*: ((*f*, *ss*), *s*) ∈ *set D* **and** *isi*: (*i*, *si*) ∈ *set* (*zip* [*0*..<*length ss*] *ss*)
from *isi* **have** *si*: *si* ∈ *set ss* **by** (*metis in-set-zipE*)
from *mem si ass*
have *si* ∈ *set S* **by** *auto*
}
thus *snd* ‘ ⋃ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat P*))) ⊆ *set S* **unfolding** *P-def pats-def* **by** *force*
qed *simp*
also have *pat-list* ‘ *set P* = { { {(*pat*,*lhs*)} | *lhs*. *lhs* ∈ *set lhss*} | *pat*. *pat* ∈ *set pats*}
unfolding *pat-list-def P-def* **by** (*auto simp*: *image-comp*)
also have *pats-complete* (*set S*) *?C* ... ⟷
Ball { *pat* · σ | *pat* σ. *pat* ∈ *set pats* ∧ *?cg-subst* (*set S*) *?C* σ} *?match-lhs* (**is** - = *Ball ?L* -)
unfolding *pattern-completeness-context.pat-complete-def*
pattern-completeness-context.match-complete-wrt-def matches-def
by *auto* (*smt* (*verit*, *best*) *case-prod-conv mem-Collect-eq singletonI*, *blast*)
also have *?L* = $\mathcal{B}$(*?C*,*?D*,∅) (**is** - = *?R*)
proof
{

fix *pat* **and** σ :: ('*f*,-,'*v*)*gsubst*
assume *pat*: *pat* $\in$ *set pats* **and** *subst*: *?cg-subst* (*set S*) *?C* σ
from *pat*[*unfolded pats-def*] **obtain** *f ss s* **where** *pat*: *pat* = *Fun f* (*map Var* (*zip* [*0..<length ss*] *ss*))
and *inDs*: ((*f*,*ss*),*s*) $\in$ *set D* **by** *auto*
from *dist*(*2*) *inDs* **have** *f*: *f* : *ss* $\rightarrow$ *s in ?D* **unfolding** *hastype-in-ssig-def* **by** *simp*
{
fix *i*
assume *i*: *i* < *length ss*
hence *ss* ! *i* $\in$ *set ss* **by** *auto*
with *inDs ass* **have** *ss* ! *i* $\in$ *set S* **by** *auto*
with *subst* **have** σ (*i*, *ss* ! *i*) : *ss* ! *i in* $\mathcal{T}$(*?C*,$\emptyset$) **unfolding** *pattern-completeness-context.cg-subst-def*

pattern-completeness-context.EMPTY-def **by** *auto*
} **note** *ssigma* = *this*
define *ts* **where** *ts* = (*map* (λ *i*. σ (*i*, *ss* ! *i*)) [*0..<length ss*])
have *ts*: *ts* :$_l$ *ss in* $\mathcal{T}$(*?C*,$\emptyset$) **unfolding** *list-all2-conv-all-nth ts-def* **using** *ssigma* **by** *auto*
have *pat*: *pat* $\cdot$ σ = *Fun f ts*
unfolding *pat ts-def* **by** (*auto intro*: *nth-equalityI*)
from *pat f ts* **have** *pat* $\cdot$ σ $\in$ *?R* **unfolding** *basic-terms-def* **by** *auto*
}
thus *?L* $\subseteq$ *?R* **by** *blast*
{
fix *f ss s* **and** *ts* :: ('*f*,'*v*)*term list*
assume *f*: *f* : *ss* $\rightarrow$ *s in ?D* **and** *ts*: *ts* :$_l$ *ss in* $\mathcal{T}$(*?C*,$\emptyset$)
from *ts* **have** *len*: *length ts* = *length ss* **by** (*metis list-all2-lengthD*)
define *pat* **where** *pat* = *Fun f* (*map Var* (*zip* [*0..<length ss*] *ss*))
from *f* **have** ((*f*,*ss*),*s*) $\in$ *set D* **unfolding** *hastype-in-ssig-def* **by** (*metis map-of-SomeD*)
hence *pat*: *pat* $\in$ *set pats* **unfolding** *pat-def pats-def* **by** *force*
define σ **where** σ *x* = (*case x of* (*i*,*s*) $\Rightarrow$ *if i* < *length ss* $\wedge$ *s* = *ss* ! *i then ts* ! *i else*
(*SOME t*. *t* : *s in* $\mathcal{T}$(*?C*,*?EMPTY*))) **for** *x*
have *id*: *Fun f ts* = *pat* $\cdot$ σ **unfolding** *pat-def* **using** *len*
by (*auto intro*!: *nth-equalityI simp*: σ*-def*)
have *ssigma*: *?cg-subst* (*set S*) *?C* σ
unfolding *pattern-completeness-context.cg-subst-def*
proof (*intro allI impI*)
fix *x* :: *nat* $\times$ -
assume *snd x* $\in$ *set S*
then obtain *i s* **where** *x*: *x* = (*i*,*s*) **and** *s*: *s* $\in$ *set S* **by** (*cases x*, *auto*)
show σ *x* : *snd x in* $\mathcal{T}$(*?C*,*?EMPTY*)
proof (*cases i* < *length ss* $\wedge$ *s* = *ss* ! *i*)
case *True*
hence *id*: σ *x* = *ts* ! *i* **unfolding** *x* σ*-def* **by** *auto*
from *ts True* **show** *?thesis* **unfolding** *id* **unfolding** *x snd-conv pattern-completeness-context.EMPTY-def*

by (*simp add*: *list-all2-conv-all-nth*)
next
case *False*
hence *id*: σ *x* = (*SOME t. t* : *s in* $\mathcal{T}$(*?C*,*?EMPTY*)) **unfolding** *x* σ*-def* **by** *auto*
from *decide-nonempty-sorts*(*1*)[*OF dist*(*1*) *refl dec*] *s*
have ∃ *t. t* : *s in* $\mathcal{T}$(*?C*,*?EMPTY*) **unfolding** *pattern-completeness-context.EMPTY-def* **by** *auto*
from *someI-ex*[*OF this*] **have** σ *x* : *s in* $\mathcal{T}$(*?C*,*?EMPTY*) **unfolding** *id* **.**
thus *?thesis* **unfolding** *x* **by** *auto*
qed
qed
from *pat id ssigma*
have *Fun f ts* ∈ *?L* **by** *auto*
}
thus *?R* ⊆ *?L* **unfolding** *basic-terms-def* **by** *auto*
qed
finally show *?thesis* **unfolding** *pat-complete-lhss-def* **by** *blast*
qed

Definition of strong quasi-reducibility and a corresponding decision procedure

definition *strong-quasi-reducible* :: (*'f*,*'s*)*ssig* ⇒ (*'f*,*'s*)*ssig* ⇒ (*'f*,*'v*)*term set* ⇒ *bool* **where**
strong-quasi-reducible C D L =
(∀ *t* ∈ $\mathcal{B}$(*C*,*D*,∅). ∃ *ti* ∈ *set* (*t* # *args t*). ∃ *l* ∈ *L*. *l matches ti*)

definition *term-and-args* :: *'f* ⇒ (*'f*,*'v*)*term list* ⇒ (*'f*,*'v*)*term list* **where**
term-and-args f ts = *Fun f ts* # *ts*

definition *decide-strong-quasi-reducible* ::
((*'f* × *'s list*) × *'s*)*list* ⇒ ((*'f* × *'s list*) × *'s*)*list* ⇒ (*'f*,*'v*)*term list* ⇒ *showsl* + *bool* **where**
decide-strong-quasi-reducible C D lhss = *do* {
check (*distinct* (*map fst C*)) (*showsl-lit* (*STR ''constructor information contains duplicate''*));
check (*distinct* (*map fst D*)) (*showsl-lit* (*STR ''defined symbol information contains duplicate''*));
let S = *sorts-of-ssig-list C*;
check-allm (λ ((*f*,*ss*),-). *check-allm* (λ *s. check* (*s* ∈ *set S*)
(*showsl-lit* (*STR ''defined symbol f has argument sort s that is not known in constructors''*))) *ss*) *D*;
(*case* (*decide-nonempty-sorts S C*) *of None* ⇒ *return* () | *Some s* ⇒ *error* (*showsl-lit* (*STR ''sort s is empty''*)));
let pats = *map* (λ ((*f*,*ss*),*s*). *term-and-args f* (*map Var* (*zip* [*0*..<*length ss*] *ss*))) *D*;
let P = *map* (*List.maps* (λ *pat. map* (λ *lhs.* [(*pat*,*lhs*)]) *lhss*)) *pats*;
return (*decide-pat-complete C P*)

}

lemma *decide-strong-quasi-reducible*:
assumes *decide-strong-quasi-reducible C D* (*lhss* :: ('*f*,'*v*)*term list*) = *return b*
shows *b* = *strong-quasi-reducible* (*map-of C*) (*map-of D*) (*set lhss*)
proof −
let *?EMPTY* = *pattern-completeness-context.EMPTY*
let *?cg-subst* = *pattern-completeness-context.cg-subst*
let *?C* = *map-of C*
let *?D* = *map-of D*
define *S* **where** *S* = *sorts-of-ssig-list C*
define *pats* **where** *pats* = *map* (λ ((*f*,*ss*),*s*). *term-and-args f* (*map Var* (*zip* [*0*..<*length ss*] *ss*))) *D*
define *P* **where** *P* = *map* (*List.maps* (λ *pat*. *map* (λ *lhs*. [(*pat*,*lhs*)]) *lhss*)) *pats*
let *?match-lhs* = λ*t*. ∃ *l* ∈ *set lhss*. *l matches t*
note *ass* = *assms*(*1*)[*unfolded decide-strong-quasi-reducible-def*, *folded S-def*, *folded pats-def*,
unfolded Let-def, *folded P-def*]
from *ass* **have** *dec*: *decide-nonempty-sorts S C* = *None* (**is** *?e* = -) **by** (*cases ?e*, *auto*)
note *ass* = *ass*[*unfolded dec*, *simplified*]
from *ass* **have** *b*: *b* = *decide-pat-complete C P* **and** *dist*: *distinct* (*map fst C*) *distinct* (*map fst D*) **by** *auto*
have *b* = *decide-pat-complete C P* **by** *fact*
also have ... = *pats-complete* (*set S*) *?C* (*pat-list* ‘ *set P*)
proof (*rule decide-pat-complete*[*OF refl dist*(*1*) *dec*[*unfolded S-def*]], *unfold S-def*[*symmetric*])
{
fix *f ss s i si*
assume *mem*: ((*f*, *ss*), *s*) ∈ *set D* **and** *isi*: (*i*, *si*) ∈ *set* (*zip* [*0*..<*length ss*] *ss*)
from *isi* **have** *si*: *si* ∈ *set ss* **by** (*metis in-set-zipE*)
from *mem si ass*
have *si* ∈ *set S* **by** *auto*
}
thus *snd* ‘ ⋃ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat P*))) ⊆ *set S* **unfolding** *P-def pats-def term-and-args-def List.maps-def*
by *fastforce*
qed *simp*
also have *pat-list* ‘ *set P* = { { {(*pat*,*lhs*)} | *lhs pat*. *pat* ∈ *set patL* ∧ *lhs* ∈ *set lhss*} | *patL*. *patL* ∈ *set pats*}
unfolding *pat-list-def P-def List.maps-def* **by** (*auto simp*: *image-comp*) *force*+
also have *pats-complete* (*set S*) *?C* ... ⟷
(∀ *patsL* σ. *patsL* ∈ *set pats* ⟶ *?cg-subst* (*set S*) *?C* σ ⟶ (∃ *pat* ∈ *set patsL*. *?match-lhs* (*pat* · σ))) (**is** - ⟷ *?L*)
unfolding *pattern-completeness-context.pat-complete-def*
pattern-completeness-context.match-complete-wrt-def matches-def
by *auto*
(*smt* (*verit*, *best*) *case-prod-conv mem-Collect-eq singletonI*,
metis (*mono-tags*, *lifting*) *case-prod-conv singleton-iff*)

also have *?L*
$\longleftrightarrow$ ($\forall$ *f ss s* (*ts* :: ($'f$,$'v$)*term list*). *f* : *ss* $\rightarrow$ *s in ?D* $\longrightarrow$ *ts* $:_l$ *ss in* $\mathcal{T}$(*?C*,$\emptyset$) $\longrightarrow$
($\exists$ *ti* $\in$ *set* (*term-and-args f ts*). *?match-lhs ti*)) (**is** - = *?R*)
proof (*standard*; *intro allI impI*)
fix *patL* **and** σ :: ($'f$,-,$'v$)*gsubst*
assume *patL*: *patL* $\in$ *set pats* **and** *subst*: *?cg-subst* (*set S*) *?C* σ **and** *R*: *?R*
from *patL*[*unfolded pats-def*] **obtain** *f ss s* **where** *patL*: *patL* = *term-and-args f* (*map Var* (*zip* [*0..<length ss*] *ss*))
and *inDs*: ((*f*,*ss*),*s*) $\in$ *set D* **by** *auto*
from *dist*(*2*) *inDs* **have** *f*: *f* : *ss* $\rightarrow$ *s in ?D* **unfolding** *hastype-in-ssig-def* **by** *simp*
{
fix *i*
assume *i*: *i* < *length ss*
hence *ss* ! *i* $\in$ *set ss* **by** *auto*
with *inDs ass* **have** *ss* ! *i* $\in$ *set S* **by** *auto*
with *subst* **have** σ (*i*, *ss* ! *i*) : *ss* ! *i in* $\mathcal{T}$(*?C*,$\emptyset$)
unfolding *pattern-completeness-context.cg-subst-def pattern-completeness-context.EMPTY-def* **by** *auto*
} **note** *ssigma* = *this*
define *ts* **where** *ts* = (*map* (λ *i*. σ (*i*, *ss* ! *i*)) [*0..<length ss*])
have *ts*: *ts* $:_l$ *ss in* $\mathcal{T}$(*?C*,$\emptyset$) **unfolding** *list-all2-conv-all-nth ts-def* **using** *ssigma* **by** *auto*
from *R*[*rule-format*, *OF f ts*] **obtain** *ti* **where** *ti*: *ti* $\in$ *set* (*term-and-args f ts*) **and** *match*: *?match-lhs ti* **by** *auto*
have *map* (λ *pat*. *pat* $\cdot$ σ) *patL* = *term-and-args f ts* **unfolding** *patL term-and-args-def ts-def*
by (*auto intro*: *nth-equalityI*)
from *ti*[*folded this*] *match*
show $\exists$*pat*$\in$*set patL*. *?match-lhs* (*pat* $\cdot$ σ) **by** *auto*
next
fix *f ss s* **and** *ts* :: ($'f$,$'v$)*term list*
assume *f*: *f* : *ss* $\rightarrow$ *s in ?D* **and** *ts*: *ts* $:_l$ *ss in* $\mathcal{T}$(*?C*,$\emptyset$) **and** *L*: *?L*
from *ts* **have** *len*: *length ts* = *length ss* **by** (*metis list-all2-lengthD*)
define *patL* **where** *patL* = *term-and-args f* (*map Var* (*zip* [*0..<length ss*] *ss*))
from *f* **have** ((*f*,*ss*),*s*) $\in$ *set D* **unfolding** *hastype-in-ssig-def* **by** (*metis map-of-SomeD*)
hence *patL*: *patL* $\in$ *set pats* **unfolding** *patL-def pats-def* **by** *force*
define σ **where** σ *x* = (*case x of* (*i*,*s*) $\Rightarrow$ *if i* < *length ss* $\wedge$ *s* = *ss* ! *i then ts* ! *i else*
(*SOME t*. *t* : *s in* $\mathcal{T}$(*?C*,*?EMPTY*))) **for** *x*
have *ssigma*: *?cg-subst* (*set S*) *?C* σ
unfolding *pattern-completeness-context.cg-subst-def*
proof (*intro allI impI*)
fix *x* :: *nat* $\times$ -
assume *snd x* $\in$ *set S*
then obtain *i s* **where** *x*: *x* = (*i*,*s*) **and** *s*: *s* $\in$ *set S* **by** (*cases x*, *auto*)
show σ *x* : *snd x in* $\mathcal{T}$(*?C*,*?EMPTY*)

proof (*cases* $i <$ *length ss* $\wedge$ $s =$ *ss* ! i)
case *True*
hence *id*: $\sigma\ x = ts\ !\ i$ **unfolding** x *σ-def* **by** *auto*
from *ts True* **show** *?thesis* **unfolding** *id* **unfolding** *x snd-conv pattern-completeness-context.EMPTY-def*
by (*simp add*: *list-all2-conv-all-nth*)
next
case *False*
hence *id*: $\sigma\ x =$ (*SOME t. t* : *s in* $\mathcal{T}$(*?C*,*?EMPTY*)) **unfolding** x *σ-def* **by** *auto*
from *decide-nonempty-sorts*(*1*)[*OF dist*(*1*) *refl dec*] *s*
have $\exists$ *t. t* : *s in* $\mathcal{T}$(*?C*,*?EMPTY*) **unfolding** *pattern-completeness-context.EMPTY-def* **by** *auto*
from *someI-ex*[*OF this*] **have** $\sigma\ x$: *s in* $\mathcal{T}$(*?C*,*?EMPTY*) **unfolding** *id* **.**
thus *?thesis* **unfolding** x **by** *auto*
qed
qed
from *L*[*rule-format*, *OF patL ssigma*]
obtain *pat* **where** *pat*: *pat* $\in$ *set patL* **and** *match*: *?match-lhs* (*pat* $\cdot$ σ) **by** *auto*
have *id*: *map* (λ *pat. pat* $\cdot$ σ) *patL* = *term-and-args f ts* **unfolding** *patL-def term-and-args-def* **using** *len*
by (*auto intro*!: *nth-equalityI simp*: *σ-def*)
show $\exists$ *ti* $\in$ *set* (*term-and-args f ts*). *?match-lhs ti* **unfolding** *id*[*symmetric*] **using** *pat match* **by** *auto*
qed
also have ... = ($\forall$ *t. t* $\in$ $\mathcal{B}$(*?C*,*?D*,$\emptyset$) $\longrightarrow$ ($\exists$ *ti* $\in$ *set* (*t* # *args t*). *?match-lhs ti*))
unfolding *basic-terms-def term-and-args-def* **by** *force*
finally show *?thesis* **unfolding** *strong-quasi-reducible-def* **by** *blast*
qed

7.1 Connecting Pattern-Completeness, Strong Quasi-Reducibility and Quasi-Reducibility

definition *quasi-reducible* :: ($'f$,$'s$)*ssig* $\Rightarrow$ ($'f$,$'s$)*ssig* $\Rightarrow$ ($'f$,$'v$)*term set* $\Rightarrow$ *bool*
where
quasi-reducible $C\ D\ L$ = ($\forall$ $t \in \mathcal{B}(C,D,\emptyset)$. $\exists$ $tp \trianglelefteq t$. $\exists\, l \in L$. *l matches tp*)

lemma *pat-complete-imp-strong-quasi-reducible*:
pat-complete-lhss $C\ D\ L \Longrightarrow$ *strong-quasi-reducible* $C\ D\ L$
unfolding *pat-complete-lhss-def strong-quasi-reducible-def* **by** *force*

lemma *arg-imp-subt*: $s \in$ *set* (*args t*) $\Longrightarrow t \trianglerighteq s$
by (*cases t*, *auto*)

lemma *strong-quasi-reducible-imp-quasi-reducible*:
strong-quasi-reducible $C\ D\ L \Longrightarrow$ *quasi-reducible* $C\ D\ L$
unfolding *strong-quasi-reducible-def quasi-reducible-def*
by (*force dest*: *arg-imp-subt*)

If no root symbol of a left-hand sides is a constructor, then pattern completeness and quasi-reducibility coincide.

lemma *quasi-reducible-iff-pat-complete*: **fixes** L :: $('f,'v)term\ set$
assumes $\bigwedge$ $l\ f\ ls\ \tau s\ \tau.\ l \in L \Longrightarrow l = Fun\ f\ ls \Longrightarrow \neg\ f : \tau s \rightarrow \tau\ in\ C$
shows *pat-complete-lhss* $C\ D\ L \longleftrightarrow$ *quasi-reducible* $C\ D\ L$
proof (*standard*, *rule strong-quasi-reducible-imp-quasi-reducible*[*OF pat-complete-imp-strong-quasi-reducible*])
assume *q*: *quasi-reducible* $C\ D\ L$
show *pat-complete-lhss* $C\ D\ L$
unfolding *pat-complete-lhss-def*
proof
fix t :: $('f,'v)term$
assume *t*: $t \in \mathcal{B}(C,D,\emptyset)$
from *q*[*unfolded quasi-reducible-def*, *rule-format*, *OF this*]
obtain *tp* **where** *tp*: $t \trianglerighteq tp$ **and** *match*: $\exists\, l \in L.\ l$ *matches tp* **by** *auto*
show $\exists\, l \in L.\ l$ *matches t*
proof (*cases* $t = tp$)
case *True*
thus *?thesis* **using** *match* **by** *auto*
next
case *False*
from *t*[*unfolded basic-terms-def*] **obtain** *f ts ss* **where** *t*: $t = Fun\ f\ ts$ **and** *ts*: $ts :_l ss\ in\ \mathcal{T}(C,\emptyset)$ **by** *auto*
from *t False tp* **obtain** *ti* **where** *ti*: $ti \in set\ ts$ **and** *subt*: $ti \trianglerighteq tp$
by (*meson Fun-supteq*)
from *subt* **obtain** *CC* **where** *ctxt*: $ti = CC\ \langle\ tp\ \rangle$ **by** *auto*
from *ti ts* **obtain** *s* **where** $ti : s\ in\ \mathcal{T}(C,\emptyset)$ **unfolding** *list-all2-conv-all-nth set-conv-nth* **by** *auto*
from *hastype-context-decompose*[*OF this*[*unfolded ctxt*]] **obtain** *s* **where** *tp*: $tp : s\ in\ \mathcal{T}(C,\emptyset)$ **by** *blast*
from *match*[*unfolded matches-def*] **obtain** $l\ \sigma$ **where** *l*: $l \in L$ **and** *match*: $tp = l \cdot \sigma$ **by** *auto*
show *?thesis*
proof (*cases l*)
case (*Var x*)
with *l* **show** *?thesis* **unfolding** *matches-def* **by** (*auto intro*!: *bexI*[*of - l*])
next
case (*Fun f ls*)
from *tp*[*unfolded match this*, *simplified*] **obtain** *ss* **where** $f : ss \rightarrow s\ in\ C$
by (*meson Fun-hastype hastype-def hastype-in-ssig-def*)
with *assms*[*OF l Fun*, *of ss s*] **show** *?thesis* **by** *auto*
qed
qed
qed
qed

end

8 Setup for Experiments

theory *Test-Pat-Complete*
  imports
    Pattern-Completeness
    HOL−Library.Code-Abstract-Char
    HOL−Library.Code-Target-Numeral
begin

turn error message into runtime error

definition *pat-complete-alg* :: $(('f \times 's\ list) \times 's)list \Rightarrow (('f \times 's\ list) \times 's)list \Rightarrow ('f,'v)term\ list \Rightarrow bool$ **where**
  pat-complete-alg C D lhss = (
  case decide-pat-complete-lhss C D lhss of Inl err ⇒ *Code.abort* (*err* (*STR* '''')) (λ -. *True*)
    | *Inr res* ⇒ *res*)

turn error message into runtime error

definition *strong-quasi-reducible-alg* :: $(('f \times 's\ list) \times 's)list \Rightarrow (('f \times 's\ list) \times 's)list \Rightarrow ('f,'v)term\ list \Rightarrow bool$ **where**
  strong-quasi-reducible-alg C D lhss = (
  case decide-strong-quasi-reducible C D lhss of Inl err ⇒ *Code.abort* (*err* (*STR* '''')) (λ -. *True*)
    | *Inr res* ⇒ *res*)

Examples

definition *nat-bool* = [
  ((''zero'', []), ''nat''),
  ((''succ'', [''nat'']), ''nat''),
  ((''true'', []), ''bool''),
  ((''false'', []), ''bool'')
  ]

definition *int-bool* = [
  ((''zero'', []), ''int''),
  ((''succ'', [''int'']), ''int''),
  ((''pred'', [''int'']), ''int''),
  ((''true'', []), ''bool''),
  ((''false'', []), ''bool'')
  ]

definition *even-nat* = [
    ((''even'', [''nat'']), ''bool'')
  ]

definition *even-int* = [
    ((''even'', [''int'']), ''bool'')
  ]

definition *even-lhss* = [
Fun ''even'' [Fun ''zero'' []],
Fun ''even'' [Fun ''succ'' [Fun ''zero'' []]],
Fun ''even'' [Fun ''succ'' [Fun ''succ'' [Var ''x'']]]
]

definition *even-lhss-int* = [
Fun ''even'' [Fun ''zero'' []],
Fun ''even'' [Fun ''succ'' [Fun ''zero'' []]],
Fun ''even'' [Fun ''succ'' [Fun ''succ'' [Var ''x'']]],
Fun ''even'' [Fun ''pred'' [Fun ''zero'' []]],
Fun ''even'' [Fun ''pred'' [Fun ''pred'' [Var ''x'']]],
Fun ''succ'' [Fun ''pred'' [Var ''x'']],
Fun ''pred'' [Fun ''succ'' [Var ''x'']]
]

lemma *decide-pat-complete-wrapper*:
assumes (*case decide-pat-complete-lhss C D lhss of Inr b* ⇒ *Some b* | *Inl -* ⇒ *None*) = *Some res*
shows *pat-complete-lhss* (*map-of C*) (*map-of D*) (*set lhss*) = *res*
using *decide-pat-complete-lhss*[*of C D lhss*] *assms* **by** (*auto split*: *sum.splits*)

lemma *decide-strong-quasi-reducible-wrapper*:
assumes (*case decide-strong-quasi-reducible C D lhss of Inr b* ⇒ *Some b* | *Inl -* ⇒ *None*) = *Some res*
shows *strong-quasi-reducible* (*map-of C*) (*map-of D*) (*set lhss*) = *res*
using *decide-strong-quasi-reducible*[*of C D lhss*] *assms* **by** (*auto split*: *sum.splits*)

lemma *pat-complete-lhss* (*map-of nat-bool*) (*map-of even-nat*) (*set even-lhss*)
apply (*subst decide-pat-complete-wrapper*[*of - - - True*])
by *eval+*

lemma ¬ *pat-complete-lhss* (*map-of int-bool*) (*map-of even-int*) (*set even-lhss-int*)

apply (*subst decide-pat-complete-wrapper*[*of - - - False*])
by *eval+*

lemma *strong-quasi-reducible* (*map-of int-bool*) (*map-of even-int*) (*set even-lhss-int*)

apply (*subst decide-strong-quasi-reducible-wrapper*[*of - - - True*])
by *eval+*

definition *non-lin-lhss* = [
Fun ''f'' [Var ''x'', Var ''x'', Var ''y''],
Fun ''f'' [Var ''x'', Var ''y'', Var ''x''],
Fun ''f'' [Var ''y'', Var ''x'', Var ''x'']
]

lemma *pat-complete-lhss* (*map-of nat-bool*) (*map-of* [((''*f*'',[''*bool*'',''*bool*'',''*bool*'']),''*bool*'')])
(*set non-lin-lhss*)
  apply (*subst decide-pat-complete-wrapper*[*of* - - - *True*])
  by *eval*+

lemma ¬ *pat-complete-lhss* (*map-of nat-bool*) (*map-of* [((''*f*'',[''*nat*'',''*nat*'',''*nat*'']),''*bool*'')])
(*set non-lin-lhss*)
  apply (*subst decide-pat-complete-wrapper*[*of* - - - *False*])
  by *eval*+

definition *testproblem* (*c* :: *nat*) *n* = (*let s* = *String.implode*; *s* = *id*;
    c1 = *even c*;
    c2 = *even* (*c div 2*);
    c3 = *even* (*c div 4*);
    c4 = *even* (*c div 8*);
    revo = (*if c4 then id else rev*);
    nn = [*0* ..< *n*];
    rnn = (*if c4 then id nn else rev nn*);
    b = *s* ''*b*''; *t* = *s* ''*tt*''; *f* = *s* ''*ff*''; *g* = *s* ''*g*'';
    gg = (λ *ts*. *Fun g* (*revo ts*));
    ff = *Fun f* [];
    tt = *Fun t* [];
    C = [((*t*, [] :: *string list*), *b*), ((*f*, []), *b*)];
    D = [((*g*, *replicate* (*2* ∗ *n*) *b*), *b*)];
    x = (λ *i* :: *nat*. *Var* (*s* (''*x*'' @ *show i*)));
    y = (λ *i* :: *nat*. *Var* (*s* (''*y*'' @ *show i*)));
    lhsF = *gg* (*if c1 then List.maps* (λ *i*. [*ff*, *y i*]) *rnn else* (*replicate n ff* @ *map*
y rnn));
    lhsT = (λ *b j*. *gg* (*if c1 then List.maps* (λ *i*. *if i* = *j then* [*tt*, *b*] *else* [*x i*, *y i*])
rnn else
            (*map* (λ *i*. *if i* = *j then tt else x i*) *rnn* @ *map* (λ *i*. *if i* = *j then b else*
y i) *rnn*)));
    lhssT = (*if c2 then List.maps* (λ *i*. [*lhsT tt i*, *lhsT ff i*]) *nn else List.maps* (λ
b. *map* (*lhsT b*) *nn*) [*tt*,*ff*]);
    lhss = (*if c3 then* [*lhsF*] @ *lhssT else lhssT* @ [*lhsF*])
  in (*C*, *D*, *lhss*))

definition *test-problem c n perms* = (*if c* < *16 then testproblem c n*
  else let (*C*, *D*, *lhss*) = *testproblem 0 n*;
      (*permRow*,*permCol*) = *perms* ! (*c* − *16*);
      permRows = *map* (λ *i*. *lhss* ! *i*) *permRow*;
      pCol = (λ *t*. *case t of Fun g ts* ⇒ *Fun g* (*map* (λ *i*. *ts* ! *i*) *permCol*))
    in (*C*, *D*, *map pCol permRows*))

definition *test-problem-integer* **where**
  test-problem-integer c n perms = *test-problem* (*nat-of-integer c*) (*nat-of-integer*
n) (*map* (*map-prod* (*map nat-of-integer*) (*map nat-of-integer*)) *perms*)

fun *term-to-haskell* **where**

term-to-haskell (*Var x*) = *String.implode x*
| *term-to-haskell* (*Fun f ts*) = (*if f* = *''tt'' then STR ''TT'' else if f* = *''ff'' then STR ''FF'' else String.implode f*)
  + *foldr* (λ *t r*. *STR* '' '' + *term-to-haskell t* + *r*) *ts* (*STR* '''')

definition *createHaskellInput* :: *integer* ⇒ *integer* ⇒ (*integer list* × *integer list*) *list* ⇒ *String.literal* **where**
  createHaskellInput c n perms = (*case test-problem-integer c n perms*
    of
    (-,-,*lhss*) ⇒ *STR ''module Test*(*g*) *where*$\boxed{\hookleftarrow}\boxed{\hookleftarrow}$*data B* = *TT* | *FF*$\boxed{\hookleftarrow}\boxed{\hookleftarrow}$''
+
      foldr (λ *l s*. (*term-to-haskell l* + *STR* '' = *TT*$\boxed{\hookleftarrow}$'' + *s*)) *lhss* (*STR* ''''))

definition *pat-complete-alg-test* :: *integer* ⇒ *integer* ⇒ (*integer list* ∗ *integer list*)*list* ⇒ *bool* **where**
  pat-complete-alg-test c n perms = (*case test-problem-integer c n perms of*
    (*C*,*D*,*lhss*) ⇒ *pat-complete-alg C D lhss*)

definition *show-pat-complete-test* :: *integer* ⇒ *integer* ⇒ (*integer list* ∗ *integer list*)*list* ⇒ *String.literal* **where**
  show-pat-complete-test c n perms = (*case test-problem-integer c n perms of* (-,-,*lhss*)

  ⇒ *showsl-lines* (*STR ''empty''*) *lhss* (*STR* ''''))

definition *create-agcp-input* :: (*String.literal* ⇒ *'t*) ⇒ *integer* ⇒ *integer* ⇒ (*integer list* ∗ *integer list*)*list* ⇒
  't list list ∗ *'t list list* **where**
  create-agcp-input term C N perms = (*let*
      n = *nat-of-integer N*;
      c = *nat-of-integer C*;
      lhss = (*snd o snd*) (*test-problem-integer C N perms*);
      tt = (λ *t*. *case t of* (*Var x*) ⇒ *term* (*String.implode* (''?'' @ *x* @ '':*B*''))
          | *Fun f* [] ⇒ *term* (*String.implode f*));
      pslist = *map* (λ *i*. *tt* (*Var* (''x'' @ *show i*))) [*0*..< *2* ∗ *n*];

      patlist = *map* (λ *t*. *case t of Fun* - *ps* ⇒ *map tt ps*) *lhss*
    in ([*pslist*], *patlist*))

connection to AGCP, which is written in SML, and SML-export of verified pattern completeness algorithm

export-code
  pat-complete-alg-test
  show-pat-complete-test
  create-agcp-input
  pat-complete-alg
  strong-quasi-reducible-alg
  Var
  in *SML* **module-name** *Pat-Complete*

tree automata encoding

We assume that there are certain interface-functions from the tree-automata library.

context
  fixes *cState* :: *String.literal* ⇒ *'state* — create a state from name
  and *cSym* :: *String.literal* ⇒ *integer* ⇒ *'sym* — create a symbol from name and arity
  and *cRule* :: *'sym* ⇒ *'state list* ⇒ *'state* ⇒ *'rule* — create a transition-rule
  and *cAut* :: *'sym list* ⇒ *'state list* ⇒ *'state list* ⇒ *'rule list* ⇒ *'aut*
    — create an automaton given the signature, the list of all states, the list of final states, and the transitions
  and *checkSubset* :: *'aut* ⇒ *'aut* ⇒ *bool* — check language inclusion
begin

we further fix the parameters to generate the example TRSs

context
  fixes *c n* :: *integer*
  and *perms* :: (*integer list* × *integer list*) *list*
begin

definition *tt* = *cSym* (*STR* ''tt'') *0*
definition *ff* = *cSym* (*STR* ''ff'') *0*
definition *g* = *cSym* (*STR* ''g'') (*2* ∗ *n*)
definition *qt* = *cState* (*STR* ''qt'')
definition *qf* = *cState* (*STR* ''qf'')
definition *qb* = *cState* (*STR* ''qb'')
definition *qfin* = *cState* (*STR* ''qFin'')
definition *tRule* = (λ *q*. *cRule tt* [] *q*)
definition *fRule* = (λ *q*. *cRule ff* [] *q*)

definition *qbRules* = [*tRule qb*, *fRule qb*]
definition *stdRules* = *qbRules* @ [*tRule qt*, *fRule qf*]
definition *leftStates* = [*qb*, *qfin*]
definition *rightStates* = [*qt*, *qf*] @ *leftStates*
definition *finStates* = [*qfin*]
definition *signature* = [*tt*, *ff*, *g*]

fun *argToState* **where**
  argToState (*Var* -) = *qb*
| *argToState* (*Fun s* []) = (*if s* = ''tt'' *then qt else if s* = ''ff'' *then qf*
    else Code.abort (*STR* ''unknown'') (λ -. *qf*))

fun *termToRule* **where**
  termToRule (*Fun* - *ts*) = *cRule g* (*map argToState ts*) *qfin*

definition *automataLeft* = *cAut signature leftStates finStates* (*cRule g* (*replicate* (*2* ∗ *nat-of-integer n*) *qb*) *qfin* # *qbRules*)
definition *automataRight* = (*case test-problem-integer c n perms of*

(-,-,*lhss*) ⇒ *cAut signature rightStates finStates* (*map termToRule lhss* @ *stdRules*))

definition *encodeAutomata* = (*automataLeft*, *automataRight*)

definition *patCompleteAutomataTest* = (*checkSubset automataLeft automataRight*)

end
end

definition *string-append* :: *String.literal* ⇒ *String.literal* ⇒ *String.literal* (**infixr** +++ *65*) **where**
string-append s t = *String.implode* (*String.explode s* @ *String.explode t*)

code-printing constant *string-append* ⇀
(*Haskell*) **infixr** *5* ++

fun *paren* **where**
paren e l r s [] = *e*
| *paren e l r s* (*x* # *xs*) = *l* +++ *x* +++ *foldr* (λ *y r*. *s* +++ *y* +++ *r*) *xs r*

definition *showAutomata* **where** *showAutomata n c perms* = (*case encodeAutomata id* (λ *n a*. *n*)
(λ *f qs q*. *paren f* (*f* +++ *STR* ''('') (*STR* '')'') (*STR* '','') *qs* +++ *STR* '' −> '' +++ *q*)
(λ *sig Q Qfin rls*.
STR ''*tree−automata has final states*: '' +++ *paren* (*STR* ''{}'') (*STR* ''{'') (*STR* ''}'') (*STR* '','') *Qfin* +++ *STR* ''$\boxed{\hookleftarrow}$''
+++ *STR* ''*and transitions*:$\boxed{\hookleftarrow}$'' +++ *paren* (*STR* '''') (*STR* '''') (*STR* '''') (*STR* ''$\boxed{\hookleftarrow}$'') *rls* +++ *STR* ''$\boxed{\hookleftarrow}\boxed{\hookleftarrow}$'') *n c perms*
of (*all*,*pats*) ⇒ *STR* ''*decide whether language of first automaton is subset of the second automaton*$\boxed{\hookleftarrow}\boxed{\hookleftarrow}$''
+++ *STR* ''*first* '' +++ *all* +++ *STR* ''$\boxed{\hookleftarrow}$*and second* '' +++ *pats*)

value *showAutomata 4 4* []

value *show-pat-complete-test 4 4* []

value *createHaskellInput 4 4* []

connection to FORT-h, generation of Haskell-examples, and Haskell tests of verified pattern completeness algorithm

export-code *encodeAutomata*
showAutomata
patCompleteAutomataTest
show-pat-complete-test
pat-complete-alg-test
createHaskellInput
in *Haskell* **module-name** *Pat-Test-Generated*

end

References

[1] T. Aoto and Y. Toyama. Ground confluence prover based on rewriting induction. In D. Kesner and B. Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, volume 52 of *LIPIcs*, pages 33:1–33:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[2] A. Lazrek, P. Lescanne, and J. Thiel. Tools for proving inductive equalities, relative completeness, and omega-completeness. *Inf. Comput.*, 84(1):47–70, 1990.

[3] A. Middeldorp, A. Lochmann, and F. Mitterwallner. First-order theory of rewriting for linear variable-separated rewrite systems: Automation, formalization, certification. *J. Autom. Reason.*, 67(2):14, 2023.

[4] R. Thiemann and A. Yamada. A verified algorithm for deciding pattern completeness. In J. Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. To appear.