Verifying a Decision Procedure for Pattern Completeness*

René Thiemann

University of Innsbruck, Austria

Akihisa Yamada

National Institute of Advanced Industrial Science and Technology, Japan

March 17, 2025

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 Auxiliary Algorithm for Testing Whether "set xs" is a Singleton Set

theory *Singleton-List*
  imports *Main*
begin

definition *singleton* $x = [x]$

fun *is-singleton-list* :: *'a list* ⇒ *bool* **where**
  is-singleton-list $[x]$ = *True*
| *is-singleton-list* $(x \# y \# xs)$ = $(x = y \wedge$ *is-singleton-list* $(x \# xs))$
| *is-singleton-list* - = *False*

lemma *is-singleton-list*: *is-singleton-list xs* ⟷ *set* (*singleton* (*hd xs*)) = *set xs*
  by (*induct xs rule*: *is-singleton-list.induct*, *auto simp*: *singleton-def*)

lemma *is-singleton-list2*: *is-singleton-list xs* ⟷ (∃ *x*. *set xs* = $\{x\}$)
  by (*induct xs rule*: *is-singleton-list.induct*, *auto*)

end

3 An Interface for Solvers for a Subset of Finite Integer Difference Logic

theory *Finite-IDL-Solver-Interface*
  imports *Main*
begin

We require a solver for (a subset of) integer-difference-logic (IDL). We basically just need comparisons of variables against constants, and difference of two variables.

Note that all variables can be assumed to be finitely bounded, so we only need a solver for finite IDL search problems. Moreover, it suffices to consider inputs where only those variables are put in comparison that share the same sort (the second parameter of a variable), and the bounds are completely determined by the sorts.

type-synonym *('v,'s)fidl-input* = *((('v* × *'s)* × *int) list* × *(('v* × *'s)* × *'v* × *'s) list list*

definition *fidl-input* :: *('v,'s)fidl-input* ⇒ *bool* **where**

fidl-input = (λ (*bnds*, *diffs*).
distinct (*map fst bnds*) ∧ (∀ *v w u*. (*v*,*w*) ∈ *set* (*concat diffs*) ⟶ *u* ∈ {*v*,*w*} ⟶ *u* ∈ *fst* ‘ *set bnds*)
∧ (∀ *v w*. (*v*,*w*) ∈ *set* (*concat diffs*) ⟶ *snd v* = *snd w*)
∧ (∀ *v w*. (*v*,*w*) ∈ *set* (*concat diffs*) ⟶ *v* ≠ *w*)
∧ (∀ *v w b1 b2*. (*v*,*b1*) ∈ *set bnds* ⟶ (*w*,*b2*) ∈ *set bnds* ⟶ *snd v* = *snd w* ⟶ *b1* = *b2*)
∧ (∀ *v b*. (*v*,*b*) ∈ *set bnds* ⟶ *b* ≥ *0*))

definition *fidl-solvable* :: (′*v*,′*s*)*fidl-input* ⇒ *bool* **where**
fidl-solvable = (λ (*bnds*, *diffs*). (∃ α :: ′*v* × ′*s* ⇒ *int*.
(∀ (*v*,*b*) ∈ *set bnds*. *0* ≤ α *v* ∧ α *v* ≤ *b*) ∧
(∀ *c* ∈ *set diffs*. ∃ (*v*,*w*) ∈ *set c*. α *v* ≠ α *w*)))

definition *finite-idl-solver* **where** *finite-idl-solver solver* = (∀ *input*.
fidl-input input ⟶ *solver input* = *fidl-solvable input*)

definition *dummy-fidl-solver* **where**
dummy-fidl-solver input = *fidl-solvable input*

lemma *dummy-fidl-solver*: *finite-idl-solver dummy-fidl-solver*
unfolding *dummy-fidl-solver-def finite-idl-solver-def* **by** *simp*

lemma *dummy-fidl-solver-code*[*code*]: *dummy-fidl-solver input* = *Code.abort* (*STR* ′′*dummy fidl solver*′′) (λ -. *dummy-fidl-solver input*)
by *simp*

end

4 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
FinFun.FinFun
begin

lemma *finite-set-Cons*:
assumes *A*: *finite A* **and** *B*: *finite B*
shows *finite* (*set-Cons A B*)

proof −
have *set-Cons A B = case-prod* (#) ‘ (*A* × *B*) **by** (*auto simp*: *set-Cons-def*)
then show *?thesis*
by (*simp add*: *finite-imageI*[*OF finite-cartesian-product*[*OF A B*],*of case-prod* (#)])
qed

lemma *finite-listset*:
assumes ∀ *A* ∈ *set As*. *finite A*
shows *finite* (*listset As*)
using *assms*
by (*induct As*) (*auto simp*: *finite-set-Cons*)

lemma *listset-conv-nth*:
xs ∈ *listset As* = (*length xs* = *length As* ∧ (∀ *i* < *length As*. *xs* ! *i* ∈ *As* ! *i*))
proof (*induct As arbitrary*: *xs*)
case (*Cons A As xs*) **then show** *?case*
by (*cases xs*) (*auto simp*: *set-Cons-def nth-Cons nat.splits*)
qed *auto*

lemma *card-listset*: **assumes** ⋀ *A*. *A* ∈ *set As* ⟹ *finite A*
shows *card* (*listset As*) = *prod-list* (*map card As*)
using *assms*
proof (*induct As*)
case (*Cons A As*)
have *sC*: *set-Cons A B = case-prod* (#) ‘ (*A* × *B*) **for** *B* **by** (*auto simp*: *set-Cons-def*)
have *IH*: *prod-list* (*map card As*) = *card* (*listset As*) **using** *Cons* **by** *auto*
have *card A* ∗ *card* (*listset As*) = *card* (*A* × *listset As*)
by (*simp add*: *card-cartesian-product*)
also have ... = *card* ((λ (*a*,*as*). *Cons a as*) ‘ (*A* × *listset As*))
by (*subst card-image*, *auto simp*: *inj-on-def*)
finally
show *?case* **by** (*simp add*: *sC IH*)
qed *auto*

4.1 Deciding the nonemptyness of all sorts under consideration

function *compute-nonempty-main* :: *'τ set* ⇒ ((*'f* × *'τ list*) × *'τ*) *list* ⇒ *'τ set*
where
compute-nonempty-main ne ls = (*let rem-ls* = *filter* (λ *f*. *snd f* ∉ *ne*) *ls in*
case partition (λ ((-,*args*),-). *set args* ⊆ *ne*) *rem-ls of*
(*new*, *rem*) ⇒ *if new* = [] *then ne else compute-nonempty-main* (*ne* ∪ *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 ... ≤ 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 × 'τ list) × 'τ) list ⇒ 'τ set where
  compute-nonempty-sorts Cs = compute-nonempty-main {} Cs

lemma compute-nonempty-sorts:
  assumes distinct (map fst Cs)
shows compute-nonempty-sorts Cs = {τ. ¬ empty-sort (map-of Cs) τ} (is - = ?NE)
proof −
  let ?TC = 𝒯(map-of Cs)
  have ne ⊆ ?NE ⟹ set ls ⊆ set Cs ⟹ snd ` (set Cs − set ls) ⊆ ne ⟹
    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 ∉ ne) ls
    have rem-ls: set rem-ls ⊆ set Cs
       snd ` (set Cs − set rem-ls) ⊆ ne
      using 1(2−) by (auto simp: rem-ls-def)
     obtain new rem where part: partition (λ((f, args), target). set args ⊆ ne)
rem-ls = (new,rem) by force
    have [simp]: compute-nonempty-main ne ls = (if new = [] then ne else com-
pute-nonempty-main (ne ∪ 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) ⊆ ?NE
    proof
      fix τ
      assume τ ∈ set (map snd new)
      then obtain f args where ((f,args),τ) ∈ set rem-ls and args: set args ⊆ ne
using part by auto
      with rem-ls have ((f,args),τ) ∈ set Cs by auto
      with assms have map-of Cs (f,args) = Some τ by auto
      hence fC: f : args → τ in map-of Cs by (simp add: fun-hastype-def)
     from args ne empty-sortI have ∀ tau. ∃ t. tau ∈ set args ⟶ t : tau in ?TC
by force
     from choice[OF this] obtain ts where ⋀ tau. tau ∈ set args ⟹ 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 τ ∈ ?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 ∪ set
(map snd new)) rem using False by simp
     also have ... = ?NE
     proof (rule IH)
       show ne ∪ set (map snd new) ⊆ ?NE using new ne by auto
       show set rem ⊆ set Cs using rem-ls part by auto
       show snd ‘ (set Cs − set rem) ⊆ ne ∪ set (map snd new)
       proof
         fix τ
         assume τ ∈ snd ‘ (set Cs − set rem)
       then obtain f args where in-ls: ((f,args),τ) ∈ set Cs and nrem: ((f,args),τ)
∉ set rem by force
         thus τ ∈ ne ∪ 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 empty-sortI obtain τ t where counter: t : τ in ?TC τ ∉ ne by
force
       thus False
       proof (induct t τ)
         case (Fun f ts τs τ)
       from Fun(1) have map-of Cs (f,τs) = Some τ by (simp add: fun-hastype-def)
         then have mem: ((f,τs),τ) ∈ set Cs by (meson map-of-SomeD)
         from Fun(3) have τs: set τs ⊆ ne by (induct, auto)
         from rem-ls mem Fun(4) have ((f,τs),τ) ∈ set rem-ls by auto
         with τs have ((f,τs),τ) ∈ 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*. *τ* ∉ *ne*) *τs*)

lemma *decide-nonempty-sorts*:
assumes *distinct* (*map fst Cs*)
shows *decide-nonempty-sorts τs Cs* = *None* ⟹ ∀ *τ* ∈ *set τs*. ¬ *empty-sort* (*map-of Cs*) *τ*
decide-nonempty-sorts τs Cs = *Some τ* ⟹ *τ* ∈ *set τs* ∧ *empty-sort* (*map-of Cs*) *τ*
unfolding *decide-nonempty-sorts-def Let-def compute-nonempty-sorts*[*OF assms*]
find-None-iff find-Some-iff **by** *auto*

4.2 Deciding infiniteness of a sort and computing cardinalities

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. Moreover, we also compute for each sort the cardinality of the set of constructor ground terms of that sort.

context
includes *finfun-syntax*
begin

fun *finfun-update-all* :: *'a list* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒*f* *'b*) ⇒ (*'a* ⇒*f* *'b*) **where**
finfun-update-all [] *g f* = *f*
| *finfun-update-all* (*x* # *xs*) *g f* = (*finfun-update-all xs g f*)(*x* $:= *g x*)

lemma *finfun-update-all*[*simp*]: *finfun-update-all xs g f* $ *x* = (*if x* ∈ *set xs then g x else f* $ *x*)
proof (*induct xs*)
case (*Cons y xs*)
thus *?case* **by** (*cases x* = *y*, *auto*)
qed *auto*

definition *compute-card-of-sort* :: *'τ* ⇒ (*'f* × *'τ list*)*list* ⇒ (*'τ* ⇒*f* *nat*) ⇒ *nat*
where
compute-card-of-sort τ cs cards = (∑ *fσs*←*remdups cs*. *prod-list* (*map* (($) *cards*) (*snd fσs*)))

function *compute-inf-card-main* :: *'τ set* ⇒ (*'τ* ⇒*f* *nat*) ⇒ (*'τ* × (*'f* × *'τ list*)*list*) *list* ⇒ *'τ set* × (*'τ* ⇒ *nat*) **where**
compute-inf-card-main m-inf cards ls = (
let (*fin*, *ls'*) =

```
      partition (λ (τ,fs). ∀ τs ∈ set (map snd fs). ∀ τ ∈ set τs. τ ∉ m-inf) ls
  in if fin = [] then (m-inf, λ τ. cards $ τ) else
    let new = map fst fin;
      cards' = finfun-update-all new (λ τ. compute-card-of-sort τ (the (map-of ls
τ)) cards) cards in
      compute-inf-card-main (m-inf − set new) cards' ls')
 by pat-completeness auto

termination
proof (relation measure (length o snd o snd), goal-cases)
 case (2 m-inf cards 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-card-main: fixes C :: ('f,'t)ssig
 assumes 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-nonempty: ∀ f τs τ τ'. f : τs → τ in C ⟶ τ' ∈ set τs ⟶ ¬
empty-sort C τ'
 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)})
 and set ls ⊆ set Cs
 and fst ` (set Cs − set ls) ∩ m-inf = {}
 and m-inf ⊆ fst ` set ls
 and ∀ τ. τ ∉ m-inf ⟶ cards $ τ = card-of-sort C τ ∧ finite-sort C τ
 and ∀ τ. τ ∈ m-inf ⟶ cards $ τ = 0
shows compute-inf-card-main m-inf cards ls = ({τ. ¬ bdd-above (size ` {t. t : τ in
𝒯(C)})},
      λ τ. card-of-sort C τ)
 using assms(7−)
proof (induct m-inf cards ls rule: compute-inf-card-main.induct)
 case (1 m-inf cards ls)
 let ?terms = λ τ. {t. t : τ in 𝒯(C)}
 let ?fin = λ τ. bdd-above (size ` ?terms τ)
 define crit where crit = (λ (τ :: 't,fs :: ('f × 't list) list). ∀ τs ∈ set (map snd
fs). ∀ τ ∈ set τs. τ ∉ m-inf)
 define S where S τ' = size ` {t. t : τ' in 𝒯(C)} for τ'
 define M where M τ' = Maximum (S τ') for τ'
 define M' where M' σs = sum-list (map M σs) + (1 + length σs) for σs
 define L where L = [ σs . (τ,cs) <− Cs, (f,σs) <− 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)
    proof −
      from inhabitet[rule-format, OF inCs] obtain f σs where (f,σs) ∈ set cs by
(cases cs,auto )
      with inCs have ((f,σs),τ) ∈ set Cs' unfolding Cs' by auto
      hence fC: f : σs → τ in C using dist(2) unfolding C-Cs
        by (meson fun-hastype-def map-of-is-SomeI)
      hence ∀σ. ∃ t. σ ∈ set σs ⟶ t : σ in 𝒯(C)
        by (auto dest!: arg-types-nonempty[rule-format] elim!: not-empty-sortE)
      from choice[OF this] obtain t where σ ∈ set σs ⟹ t σ : σ in 𝒯(C) for σ
by auto
      hence Fun f (map t σs) : τ in 𝒯(C) 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

  define cards' where cards' = finfun-update-all (map fst fin) (λ τ. compute-card-of-sort
τ (the (map-of ls τ)) cards) cards
  {
    fix τ
    assume asm: τ ∈ fst ‘ set fin
    let ?TT = ?terms τ
    from asm obtain cs where tau-cs-fin: (τ,cs) ∈ set fin by auto
    hence tau-ls: (τ,cs) ∈ set ls using part by auto
    with dist(1) ‹set ls ⊆ set Cs›
    have map: map-of Cs τ = Some cs map-of ls τ = Some cs
     by (metis (no-types, opaque-lifting) eq-key-imp-eq-value map-of-SomeD subsetD
weak-map-of-SomeI)+
    from asm have cards': cards' $ τ = compute-card-of-sort τ cs cards unfolding
cards'-def by (auto simp: map)
    from part asm have tau-fin: τ ∈ set (map fst fin) by auto
    {
      fix f σs
      have f : σs → τ in C ⟷ ((f,σs),τ) ∈ set Cs'
      proof
        assume f : σs → τ in C
       hence map-of Cs' (f,σs) = Some τ unfolding C-Cs by (rule fun-hastypeD)
        thus ((f,σs),τ) ∈ set Cs' by (rule map-of-SomeD)
      next
        assume ((f, σs), τ) ∈ set Cs'
        hence map-of Cs' (f, σs) = Some τ using dist(2) by simp
        thus f : σs → τ in C unfolding C-Cs by (rule fun-hastypeI)
      qed
      also have ... ⟷ (∃ cs. (τ, cs) ∈ set Cs ∧ (f,σs) ∈ set cs)
        unfolding Cs' by auto
      also have ... ⟷ (∃ cs. map-of Cs τ = Some cs ∧ (f,σs) ∈ set cs)
        using dist(1) by simp
      also have ... ⟷ (f,σs) ∈ set cs unfolding map by auto
```

```
    finally have (f : σs → τ in C) = ((f, σs) ∈ set cs) by auto
  } note C-to-cs = this

  define T where T σ = ?terms σ for σ
  have to-ls: {ts. ts :l σs in 𝒯(C)} = listset (map T σs) for σs
  by (intro set-eqI, unfold listset-conv-nth, auto simp: T-def list-all2-conv-all-nth)
  {
    fix f σs σ
    assume in-cs: (f, σs) ∈ set cs σ ∈ set σs
    from tau-cs-fin part have crit (τ,cs) by auto
    from this[unfolded crit-def split] in-cs have σ ∉ m-inf by auto
    with 1(6) have cards $ σ = card (T σ) and finite (T σ)
      by (auto simp: T-def card-of-sort finite-sort)
  } note σs-infos = this

  have ?TT = { Fun f ts | f ts σs. f : σs → τ in C ∧ ts :l σs in 𝒯(C)} (is - =
?FunApps)
  proof (intro set-eqI)
    fix t
    {
      assume t : τ in 𝒯(C)
      hence t ∈ ?FunApps by (induct, auto)
    }
    moreover
    {
      assume t ∈ ?FunApps
      hence t : τ in 𝒯(C) by (auto intro: Fun-hastypeI)
    }
    ultimately show t ∈ ?TT ⟷ t ∈ ?FunApps by auto
  qed
   also have ... = { Fun f ts | f ts σs. (f, σs) ∈ set cs ∧ ts :l σs in 𝒯(C)}
unfolding C-to-cs ..
  also have ... = (λ (f, ts). Fun f ts) ` (⋃ (f, σs) ∈ set cs. Pair f ` { ts. ts :l σs
in 𝒯(C)}) (is - = ?f ` ?A) by auto
  finally have TTfA: ?TT = ?f ` ?A .
 have finPair: finite (Pair f ` A) = finite A for f :: 'f and A :: ('f, 'v) Term.term
list set
    by (intro finite-image-iff inj-onI, auto)
  have inj: inj ?f by (intro injI, auto)
  from inj have card: card ?TT = card ?A
    unfolding TTfA by (meson UNIV-I card-image inj-on-def)
  also have ... = (∑ i∈set cs. card (case i of (f, σs) ⇒ Pair f ` listset (map T
σs))) unfolding to-ls
    proof (rule card-UN-disjoint[OF finite-set ballI ballI[OF ballI[OF impI]]],
goal-cases)
    case ∗: (1 fσs)
    obtain f σs where fσs: fσs = (f,σs) by force
      thus ?case using ∗ σs-infos(2) by (cases fσs, auto intro!: finite-imageI
finite-listset)
```

```
    next
      case ∗: (2 fσs gτs)
      obtain f σs where fσs: fσs = (f,σs) by force
      obtain g τs where gτs: gτs = (g,τs) by force
      show ?case
      proof (cases g = f)
        case False
        thus ?thesis unfolding fσs gτs split by auto
      next
        case True
        note fτs = gτs[unfolded True]
        show ?thesis
        proof (rule ccontr)
          assume ¬ ?thesis
          from this[unfolded fσs fτs split]
          obtain ts where ts: ts ∈ listset (map T σs) ts ∈ listset (map T τs) by
auto
            hence len: length σs = length ts length τs = length ts unfolding list-
set-conv-nth by auto
          from ∗(3)[unfolded fσs fτs] have σs ≠ τs by auto
          with len obtain i where i: i < length ts and diff: σs ! i ≠ τs ! i
            by (metis nth-equalityI)
          define ti where ti = ts ! i
          define σi where σi = σs ! i
          define τi where τi = τs ! i
          note diff = diff[folded σi-def τi-def]
          from ts i have ti ∈ T σi ti ∈ T τi
            unfolding ti-def σi-def τi-def listset-conv-nth by auto
          hence ti: ti : σi in T(C) ti : τi in T(C) unfolding T-def by auto
          hence σi = τi by fastforce
          with diff show False ..
        qed
      qed
    qed
    also have ... = (∑ fσs ∈set cs. card (listset (map T (snd fσs))))
    proof (rule sum.cong[OF refl], goal-cases)
      case (1 fσs)
      obtain f σs where id: fσs = (f,σs) by force
      show ?case unfolding id split snd-conv
        by (rule card-image, auto simp: inj-on-def)
    qed
    also have ... = (∑ fσs ∈set cs. prod-list (map card (map T (snd fσs))))
      by (rule sum.cong[OF refl], rule card-listset, insert σs-infos, auto)
    also have ... = (∑ fσs ∈set cs. prod-list (map (($) cards) (snd fσs)))
      unfolding map-map o-def using σs-infos
      by (intro sum.cong[OF refl] arg-cong[of - - prod-list], auto)
    also have ... = sum-list (map (λ fσs. prod-list (map (($) cards) (snd fσs)))
(remdups cs))
      by (rule sum.set-conv-list)
```

```
    also have ... = cards' $ τ unfolding cards' compute-card-of-sort-def ..
    finally have cards': card ?TT = cards' $ τ by auto


    from inj have finite ?TT = finite ?A
      by (metis (no-types, lifting) TTfA finite-imageD finite-imageI subset-UNIV
subset-inj-on)
     also have ... = (∀ f σs. (f,σs) ∈ set cs ⟶ finite (Pair f ‘ {ts. ts :ₗ σs in
𝒯(C)}))
      by auto
     finally have finite ?TT = (∀ f σs. (f,σs) ∈ set cs ⟶ finite {ts. ts :ₗ σs in
𝒯(C)})
      unfolding finPair by auto
     also have ... = True unfolding to-ls using σs-infos(2) by (auto intro!:
finite-listset)
    finally have fin: finite ?TT by simp

    from fin cards'
    have cards' $ τ = card (?terms τ) finite (?terms τ) ?fin τ by auto
  } note fin = this

  show ?case
  proof (cases fin = [])
    case False
     hence compute-inf-card-main m-inf cards ls = compute-inf-card-main (m-inf
− set (map fst fin)) cards' ls'
       unfolding compute-inf-card-main.simps[of m-inf] part[unfolded crit-def]
cards'-def Let-def by auto
    also have ... = ({τ. ¬ ?fin τ}, λ τ. card-of-sort C 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 m-inf − set (map fst fin) ⊆ fst ‘ set ls' using 1(5) part by force
     show ∀τ. τ ∉ m-inf − set (map fst fin) ⟶ cards' $ τ = card-of-sort C τ ∧
finite-sort C τ
      proof (intro allI impI)
       fix τ
       assume nmem: τ ∉ m-inf − set (map fst fin)
       show cards' $ τ = card-of-sort C τ ∧ finite-sort C τ
       proof (cases τ ∈ set (map fst fin))
         case False
         with nmem have tau: τ ∉ m-inf by auto
         with False 1(6)[rule-format, OF this] show ?thesis
           unfolding cards'-def by auto
       next
         case True
         with fin show ?thesis by (auto simp: card-of-sort finite-sort)
       qed
```

```
    qed
    thus ∀τ. τ ∉ m-inf − set (map fst fin) ⟶ ?fin τ
      by (force simp: 1(2) intro: fin(3))
     show ∀τ. τ ∈ m-inf − set (map fst fin) ⟶ cards' $ τ = 0 using 1(7)
unfolding cards'-def
      by auto
   qed (auto simp: cards'-def)
   finally show ?thesis .
 next
   case True
   let ?cards = λτ. cards $ τ
   have m-inf: m-inf = {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 fun-hastype-def
.
     have ∀ σ. ∃ t. σ ∈ set τs ⟶ t : σ in T(C)
    by (auto dest!: arg-types-nonempty[rule-format, OF c] elim!: not-empty-sortE)
     from choice[OF this] obtain t where ⋀ σ. σ ∈ set τs ⟹ t σ : σ in T(C)
by auto
     hence list: map t τs :l τs in T(C) by (simp add: list-all2-conv-all-nth)
     with c have Fun c (map t τs) : τ in T(C) by (intro Fun-hastypeI)
     with ∗ c list have ∃ c τs τ' ts. Fun c ts : τ in T(C) ∧ ts :l τs in T(C) ∧
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 T(C) ∧ 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 T(C) c : τs → τ in C τ' ∈ set τs τ' ∈ 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 𝒯(C) and len: length
ts = length τs by auto
        from Suc(1)[OF *(4)] obtain t where t:t : τ' in 𝒯(C) and ns:n ≤ size
t by auto
        define ts' where ts' = ts[i := t]
        have ts' :ₗ τs in 𝒯(C) 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 𝒯(C) apply (intro Fun-hastypeI[OF *(2)]) by
fastforce
        have t ∈ set ts' unfolding ts'-def using t
          by (simp add: i len set-update-memI)
        hence size (Fun c ts') ≥ Suc n using *
          by (simp add: size-list-estimation' ns)
        thus ?case using ** by blast
       qed
     } note main = this
     show m-inf ⊆ {τ. ¬ ?fin τ}
     proof (standard, standard)
       fix τ
       assume asm: τ ∈ m-inf
       have ∃ t. t : τ in 𝒯(C) ∧ n < size t for n using main[OF asm, of Suc n]
by auto
       thus ¬ ?fin τ
         by (metis bdd-above-Maximum-nat imageI mem-Collect-eq order.strict-iff)
     qed
   qed
   from True have compute-inf-card-main m-inf cards ls = (m-inf, ?cards)
     unfolding compute-inf-card-main.simps[of m-inf] part[unfolded crit-def] by
auto
   also have ?cards = (λ τ. card-of-sort C τ)
   proof (intro ext)
     fix τ
     show cards $ τ = card-of-sort C τ
     proof (cases τ ∈ m-inf)
       case False
       thus ?thesis using 1(6) by auto
     next
       case True
       define TT where TT = ?terms τ
      from True m-inf have ¬ bdd-above (size ' TT) unfolding TT-def by auto
       hence infinite TT by auto
       hence card TT = 0 by auto
     thus ?thesis unfolding TT-def using True 1(7) by (auto simp: card-of-sort)
     qed
   qed
   finally show ?thesis using m-inf by auto
 qed
```

qed

definition *compute-inf-card-sorts* :: $((('f \times 't\ list) \times 't)list \Rightarrow 't\ set \times ('t \Rightarrow nat)$
where
compute-inf-card-sorts Cs = (*let*
Cs′ = *map* (λ τ. (τ, *map fst* (*filter*(λ*f*. *snd f* = τ) *Cs*))) (*remdups* (*map snd Cs*))
in compute-inf-card-main (*set* (*map fst Cs′*)) (*K$ 0*) *Cs′*)

lemma *finite-imp-size-bdd-above*: **assumes** *finite T*
shows *bdd-above* (*size ‘ T*)
proof −
from *assms* **have** *finite* (*size ‘ T*) **by** *auto*
thus *?thesis* **by** *simp*
qed

lemma *finite-sig-imp-finite-terms-of-bounded-size*: **assumes** *finite* (*dom F*) **and** *finite* (*dom V*)
shows *finite* {*t*. ∃ τ. *size t* ≤ *n* ∧ *t* : τ *in* $\mathcal{T}(F,V)$} (**is** *finite* (*?terms n*))
proof (*induct n*)
case (*0*)
have *t* ∉ *?terms 0* **for** *t* **by** (*cases t*, *auto*)
hence *id*: *?terms 0* = {} **by** *auto*
show *?case* **unfolding** *id* **by** *simp*
next
case (*Suc n*)
let *?funsInter* = (λ (*f*, τ*s*). (*f*, *listset* (*map* (λ -. (*?terms n*)) τ*s*))) ‘ *dom F*
define *funsI* **where** *funsI* = *?funsInter*
let *?funs* = ⋃ ((λ (*f*, *tss*). *Fun f ‘ tss*) ‘ *funsI*)
{
fix *t*
assume *t* ∈ *?terms* (*Suc n*)
then obtain τ **where** *tτ*: *t* : τ *in* $\mathcal{T}(F,V)$ **and** *size*: *size t* ≤ *Suc n* **by** *auto*
have *t* ∈ *Var ‘ dom V* ∪ *?funs*
proof (*cases t*)
case (*Var x*)
thus *?thesis* **using** *tτ* **by** *auto*
next
case *t*: (*Fun f ts*)
from *tτ*[*unfolded t Fun-hastype*] **obtain** τ*s* **where** *ts*: *ts* :$_l$ τ*s in* $\mathcal{T}(F,V)$
and *f*: (*f*,τ*s*) ∈ *dom F* **by** *auto*
hence (*f*, *listset* (*map* (λ -. (*?terms n*)) τ*s*)) ∈ *funsI* **unfolding** *funsI-def* **by** *auto*
moreover have *ts* ∈ *listset* (*map* (λ -. (*?terms n*)) τ*s*)
unfolding *listset-conv-nth length-map*
proof (*intro conjI allI impI*)
show *len*: *length ts* = *length* τ*s* **using** *ts* **by** (*metis list-all2-lengthD*)
fix *i*

```
      assume i: i < length τs
      with ts have i': i < length ts and type: ts ! i : τs ! i in 𝒯(F,V)
        using list-all2-nthD2[OF ts] len by auto
      from i' have ts ! i ∈ set ts by auto
       from split-list[OF this] obtain bef aft where ts = bef @ ts ! i # aft by
auto
       from size[unfolded t] this have size (Fun f (bef @ ts ! i # aft)) ≤ Suc n
by simp
      hence size (ts ! i) ≤ n by simp
      with type have ts ! i ∈ ?terms n by auto
      with i show ts ! i ∈ map (λ-. ?terms n) τs ! i by auto
     qed
     ultimately show ?thesis unfolding t by blast
   qed
 }
 hence ?terms (Suc n) ⊆ Var ‘ dom V ∪ ?funs by blast
 moreover have finite (Var ‘ dom V ∪ ?funs)
 proof (intro finite-UnI finite-imageI assms finite-Union)
   show finite (funsI) unfolding funsI-def
     by (intro finite-imageI assms)
   fix M
   assume M ∈ {Fun f ‘ tss |. (f, tss) ∈ funsI}
    from this obtain f tss where tss: (f, tss) ∈ funsI and M: M = Fun f ‘ tss
by auto
    from tss[unfolded funsI-def] obtain τs where
     tss: tss = listset (map (λ-. {t. size t ≤ n ∧ (∃τ. t : τ in 𝒯(F,V))}) τs) and
τs ∈ snd ‘ dom F
     by force
   have finite tss unfolding tss
     by (intro finite-listset, insert Suc, auto)
   thus finite M unfolding M
     by (intro finite-imageI)
 qed
 ultimately show ?case by (rule finite-subset)
qed

lemma finite-sig-bdd-above-imp-finite: assumes finite (dom F) and finite (dom
V)
 and bdd-above (size ‘ {t. t : τ in 𝒯(F,V)})
shows finite {t. t : τ in 𝒯(F,V)}
proof −
 from assms(3)[unfolded bdd-above-def] obtain n where
   size: ∀ s∈size ‘ {t. t : τ in 𝒯(F,V)}. s ≤ n by auto
 from finite-sig-imp-finite-terms-of-bounded-size[OF assms(1−2)]
 have fin: finite {t. ∃τ. size t ≤ n ∧ t : τ in 𝒯(F,V)} by auto
 have finite {t. size t ≤ n ∧ t : τ in 𝒯(F,V)}
   by (rule finite-subset[OF - fin], auto)
 also have {t. size t ≤ n ∧ t : τ in 𝒯(F,V)} = {t. t : τ in 𝒯(F,V)}
   using size by blast
```

finally show *?thesis* **by** *auto*
qed

lemma *finite-sig-bdd-above-iff-finite*: **assumes** *finite* (*dom F*) **and** *finite* (*dom V*)

shows *bdd-above* (*size* ‘ {*t*. *t* : *τ* *in* $\mathcal{T}$(*F*,*V*)}) = *finite* {*t*. *t* : *τ* *in* $\mathcal{T}$(*F*,*V*)}
using *finite-sig-bdd-above-imp-finite*[*OF assms*] *finite-imp-size-bdd-above*
by *metis*

lemma *compute-inf-card-sorts*:
fixes *C* :: (*'f*,*'t*)*ssig*
assumes *C-Cs*: *C* = *map-of Cs*
and *arg-types-nonempty*: ∀ *f τs τ τ'*. *f* : *τs* → *τ* *in* *C* ⟶ *τ'* ∈ *set τs* ⟶ ¬ *empty-sort C τ'*
and *dist*: *distinct* (*map fst Cs*)
and *result*: *compute-inf-card-sorts Cs* = (*unb*, *cards*)
shows *unb* = {*τ*. ¬ *bdd-above* (*size* ‘ {*t*. *t* : *τ* *in* $\mathcal{T}$(*C*)})} (**is** - = *?unb*)
cards = *card-of-sort C* (**is** - = *?cards*)
unb = {*τ*. ¬*finite-sort C τ*} (**is** - = *?inf*)
proof −
let *?terms* = *λ τ*. {*t*. *t* : *τ* *in* $\mathcal{T}$(*C*)}
define *taus* **where** *taus* = *remdups* (*map snd Cs*)
define *Cs'* **where** *Cs'* = *map* (*λ τ*. (*τ*, *map fst* (*filter*(*λf*. *snd f* = *τ*) *Cs*))) *taus*
have *compute-inf-card-sorts Cs* = *compute-inf-card-main* (*set* (*map fst Cs'*)) (*K$ 0*) *Cs'*
unfolding *compute-inf-card-sorts-def taus-def Cs'-def Let-def* **by** *auto*
also have ... = (*?unb*, *?cards*)
proof (*rule compute-inf-card-main*[*OF C-Cs - arg-types-nonempty - 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* (*λ*(*τ*, *fs*). *map* (*λf*. (*f*, *τ*)) *fs*) *Cs'*))
unfolding *Cs'-def taus-def* **by** *force*
show ∀ *τ fs*. (*τ*, *fs*) ∈ *set Cs'* ⟶ *set fs* ≠ {}
unfolding *Cs'-def taus-def* **by** (*force simp*: *filter-empty-conv*)
show *fst* ‘ (*set Cs'* − *set Cs'*) ∩ *set* (*map fst Cs'*) = {} **by** *auto*
show *set* (*map fst Cs'*) ⊆ *fst* ‘ *set Cs'* **by** *auto*
{ fix *τ*
assume *τ* ∉ *set* (*map fst Cs'*)
hence *τ* ∉ *snd* ‘ *set Cs* **unfolding** *Cs'-def taus-def* **by** *auto*
hence *diff*: *C f* ≠ *Some τ* **for** *f* **unfolding** *C-Cs*
by (*metis Some-eq-map-of-iff dist imageI snd-conv*)
have *emp*: *empty-sort C τ*
proof (*intro empty-sortI notI*)
fix *t*
assume *t* : *τ* *in* $\mathcal{T}$(*C*)
thus *False* **using** *diff*
proof *induct*

```
        case (Fun f ss σs τ)
        from Fun(1,4) show False unfolding fun-hastype-def by auto
      qed auto
    qed
  }
  note ∗ = this
  show ∀τ. τ ∉ set (map fst Cs') ⟶ bdd-above (size ‘ ?terms τ)
    ∀τ. τ ∉ set (map fst Cs') ⟶ (K$ 0) $ τ = card-of-sort C τ ∧ finite-sort C
τ
    by (auto simp del: set-map dest!: ∗)
 qed auto
 finally show unb: unb = ?unb and cards: cards = ?cards unfolding result by
auto
 show unb = ?inf unfolding unb
 proof (subst finite-sig-bdd-above-iff-finite)
  show finite (dom C) unfolding C-Cs by (rule finite-dom-map-of)
  show finite (dom ∅) by auto
 qed (auto simp: finite-sort)
qed
end

abbreviation compute-inf-sorts :: (('f × 't list) × 't)list ⇒ 't set where
 compute-inf-sorts Cs ≡ fst (compute-inf-card-sorts Cs)

lemma compute-inf-sorts:
 assumes arg-types-nonempty: ∀ f τs τ τ'. f : τs → τ in map-of Cs ⟶ τ' ∈ set
τs ⟶ ¬ empty-sort (map-of Cs) τ'
 and dist: distinct (map fst Cs)
shows
 compute-inf-sorts Cs = {τ. ¬ bdd-above (size ‘ {t. t : τ in T(map-of Cs)})}
 compute-inf-sorts Cs = {τ. ¬ finite-sort (map-of Cs) τ}
 using compute-inf-card-sorts[OF refl assms]
  by (cases compute-inf-card-sorts Cs, auto)+

end
```

5 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.

6 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
    Complete-Non-Orders.Complete-Relations
    Sorted-Terms.Sorted-Contexts
    Compute-Nonempty-Infinite-Sorts
begin

lemmas *type-conversion* = *hastype-in-Term-empty-imp-subst*

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*: $f\ y = f\ z \Longrightarrow Bex\ (insert\ y\ A)\ f = Bex\ (insert\ z\ A)\ f$
  by *auto*

lemma *not-bdd-above-natD*:
  assumes $\neg$ *bdd-above* $(A :: nat\ set)$
  shows $\exists\ x \in A.\ x > n$
  using *assms* **by** (*meson bdd-above.unfold linorder-le-cases order.strict-iff*)

lemma *list-eq-nth-eq*: $xs = ys \longleftrightarrow length\ xs = length\ ys \wedge (\forall\ i < length\ ys.\ xs\ !\ i = ys\ !\ i)$
  using *nth-equalityI* **by** *metis*

lemma *subt-size*: $p \in poss\ t \Longrightarrow size\ (t\ |\text{-}\ p) \leq size\ t$
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*: $t\ |\text{-}\ (i\ \#\ p) = (ss\ !\ i)\ |\text{-}\ p$
      and $p \in poss\ (ss\ !\ i)$ **by** *auto*
    with *Cons*(*1*)[*OF this*(*3*)]
    have $size\ (t\ |\text{-}\ (i\ \#\ p)) \leq size\ (ss\ !\ i)$ **by** *auto*
   also have $\ldots \leq size\ t$ **using** *i* **unfolding** *Fun* **by** (*simp add*: *termination-simp*)
    finally show *?thesis* **.**
  qed *auto*
qed *auto*

lemma *removeAll-remdups*: *removeAll x* (*remdups ys*) = *remdups* (*removeAll x ys*)
by (*simp add*: *remdups-filter removeAll-filter-not-eq*)

lemma *removeAll-eq-Nil-iff*: *removeAll x ys* = [] ⟷ (∀ *y* ∈ *set ys*. *y* = *x*)
by (*induction ys*, *auto*)

lemma *concat-removeAll-Nil*: *concat* (*removeAll* [] *xss*) = *concat xss*
by (*induction xss*, *auto*)

lemma *removeAll-eq-imp-concat-eq*:
assumes *removeAll* [] *xss* = *removeAll* [] *xss′*
shows *concat xss* = *concat xss′*
apply (*subst* (*1 2*) *concat-removeAll-Nil*[*symmetric*])
by (*simp add*: *assms*)

lemma *map-remdups-commute*:
assumes *inj-on f* (*set xs*)
shows *map f* (*remdups xs*) = *remdups* (*map f xs*)
using *assms* **by** (*induction xs*, *auto*)

lemma *Uniq-False*: ∃$_{\leq 1}$ *a*. *False* **by** (*auto intro*!: *Uniq-I*)

abbreviation *UNIQ A* ≡ ∃$_{\leq 1}$ *a*. *a* ∈ *A*

lemma *Uniq-eq-the-elem*:
assumes *UNIQ A* **and** *a* ∈ *A* **shows** *a* = *the-elem A*
using *the1-equality′*[*OF assms*]
by (*metis assms empty-iff is-singletonI′ is-singleton-some-elem some-elem-nonempty the1-equality′ the-elem-eq*)

lemma *bij-betw-imp-Uniq-iff*:
assumes *bij-betw f A B* **shows** *UNIQ A* ⟷ *UNIQ B*
using *assms*[*THEN bij-betw-imp-surj-on*]
apply (*auto simp*: *Uniq-def*)
by (*metis assms bij-betw-def imageI inv-into-f-eq*)

lemma *image-Uniq*: *UNIQ A* ⟹ *UNIQ* (*f* ‘ *A*)
by (*smt* (*verit*) *Uniq-I image-iff the1-equality′*)

lemma *successively-eq-iff-Uniq*: *successively* (=) *xs* ⟷ *UNIQ* (*set xs*) (**is** *?l* ⟷ *?r*)
proof
show *?l* ⟹ *?r*
apply (*induction xs rule*: *induct-list012*)
by (*auto intro*: *Uniq-I*)
show *?r* ⟹ *?l*
proof (*induction xs*)

```
    case Nil
    then show ?case by simp
  next
    case xxs: (Cons x xs)
    show ?case
    proof (cases xs)
      case Nil
      then show ?thesis by simp
    next
      case xs: (Cons y ys)
      have successively (=) xs
        apply (rule xxs(1)) using xxs(2) by (simp add: Uniq-def)
      with xxs(2)
      show ?thesis by (auto simp: xs Uniq-def)
    qed
  qed
qed
```

6.1 Defining Pattern Completeness

We first consider matching problems, which are set of matching atoms. Each matching atom is a pair of terms: matchee and pattern. Matchee and pattern may have different type of variables: Matchees use natural numbers (annotated with sorts) as variables, so that it is easy to generate new variables, whereas patterns allow 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.

```
abbreviation tvars :: nat × 's ⇀ 's (𝒱) where 𝒱 ≡ sort-annotated

type-synonym ('f,'v,'s)match-atom = ('f,nat × 's)term × ('f,'v)term
type-synonym ('f,'v,'s)match-problem-set = ('f,'v,'s) match-atom 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 ≡ {{}}

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

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

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

definition *subst-left* :: ('*f*,*nat* × '*s*)*subst* ⇒ (('*f*,*nat* × '*s*)*term* × ('*f*,'*v*)*term*) ⇒ (('*f*,*nat* × '*s*)*term* × ('*f*,'*v*)*term*) **where**
subst-left τ = (λ(*t*,*r*). (*t* · τ, *r*))

A definition of pattern completeness for pattern problems.

definition *match-complete-wrt* :: ('*f*,*nat* × '*s*,'*w*)*gsubst* ⇒ ('*f*,'*v*,'*s*)*match-problem-set* ⇒ *bool* **where**
match-complete-wrt σ *mp* = (∃ μ. ∀ (*t*,*l*) ∈ *mp*. *t* · σ = *l* · μ)

lemma *match-complete-wrt-cong*:
assumes *s*: ⋀*x*. *x* ∈ *tvars-match mp* ⟹ σ *x* = σ′ *x*
and *mp*: *mp* = *mp*′
shows *match-complete-wrt* σ *mp* = *match-complete-wrt* σ′ *mp*′
apply (*unfold match-complete-wrt-def Ball-Pair-conv mp*[*symmetric*])
apply (*intro ex-cong1 all-cong1 imp-cong*[*OF refl*])
proof−
fix μ *t l* **assume** (*t*,*l*) ∈ *mp*
with *s* **have** ∀ *x* ∈ *vars t*. σ *x* = σ′ *x* **by** (*auto simp*: *tvars-match-def*)
from *subst-same-vars*[*OF this*] **show** *t*·σ = *l*·μ ⟷ *t*·σ′ = *l*·μ **by** *simp*
qed

lemma *match-complete-wrt-imp-o*:
assumes *match-complete-wrt* σ *mp* **shows** *match-complete-wrt* (σ ∘$_s$ τ) *mp*
proof (*unfold match-complete-wrt-def*)
from *assms*[*unfolded match-complete-wrt-def*] **obtain** μ **where** *eq*: ∀ (*t*,*l*) ∈ *mp*. *t*·σ = *l*·μ
by *auto*
{ fix *t l*
assume *tl*: (*t*,*l*) ∈ *mp*
with *eq* **have** *t*·(σ ∘$_s$ τ) = *l*·(μ ∘$_s$ τ) **by** *auto*
}
then show ∃ μ′. ∀ (*t*,*l*) ∈ *mp*. *t*·(σ ∘$_s$ τ) = *l*·μ′ **by** *blast*
qed

lemma *match-complete-wrt-o-imp*:
assumes *s*: σ :$_s$ $\mathcal{V}$ |‘ *tvars-match mp* → $\mathcal{T}$(*C*,∅) **and** *m*: *match-complete-wrt* (σ ∘$_s$ τ) *mp*
shows *match-complete-wrt* σ *mp*
proof (*unfold match-complete-wrt-def*)
from *m*[*unfolded match-complete-wrt-def*] **obtain** μ **where** *eq*: ∀ (*t*,*l*) ∈ *mp*. *t*·σ·τ = *l*·μ
by *auto*
have ∀ *x* ∈ *tvars-match mp*. σ *x* : *snd x in* $\mathcal{T}$(*C*,∅)
by (*auto intro*!: *sorted-mapD*[*OF s*] *simp*: *hastype-restrict*)
then have *g*: *x* ∈ *tvars-match mp* ⟹ *ground* (σ *x*) **for** *x*
by (*auto simp*: *hatype-imp-ground*)
{ fix *t l*
assume *tl*: (*t*,*l*) ∈ *mp*
then have *ground* (*t*·σ) **by** (*force intro*!: *g simp*: *tvars-match-def*)

then have $t{\cdot}\sigma{\cdot}\tau{\cdot}undefined = t{\cdot}\sigma$ **by** (*metis eval-subst ground-subst-apply*)
with *tl eq* **have** $t{\cdot}\sigma = l{\cdot}(\mu \circ_s undefined)$ **by** *auto*
}
then show $\exists\,\mu'.\ \forall\,(t,l) \in mp.\ t{\cdot}\sigma = l{\cdot}\mu'$ **by** *blast*
qed

Pattern completeness is match completeness w.r.t. any constructor-ground substitution. Note that variables to instantiate are represented as pairs of (number, sort).

definition *pat-complete* :: $('f,'s)$ *ssig* $\Rightarrow$ $('f,'v,'s)$*pat-problem-set* $\Rightarrow$ *bool* **where**
pat-complete C pp $\longleftrightarrow$ $(\forall\,\sigma :_s \mathcal{V}\ |`$ *tvars-pat pp* $\rightarrow \mathcal{T}(C).\ \exists$ *mp* $\in$ *pp. match-complete-wrt* σ *mp*)

lemma *pat-completeD*:
assumes *pp*: *pat-complete C pp*
and *s*: $\sigma :_s \mathcal{V}\ |`$ *tvars-pat pp* $\rightarrow \mathcal{T}(C,\emptyset)$
shows $\exists$ *mp* $\in$ *pp. match-complete-wrt* σ *mp*
proof −
from *s* **have** $\sigma \circ_s undefined :_s \mathcal{V}\ |`$ *tvars-pat pp* $\rightarrow \mathcal{T}(C)$
by (*simp add*: *subst-compose-sorted-map*)
from *pp*[*unfolded pat-complete-def*, *rule-format*, *OF this*]
obtain *mp* **where** *mp*: *mp* $\in$ *pp*
and *m*: *match-complete-wrt* ($\sigma \circ_s$ *undefined* :: - $\Rightarrow$ (-,*unit*) *term*) *mp*
by *auto*
have $\sigma :_s \mathcal{V}\ |`$ *tvars-match mp* $\rightarrow \mathcal{T}(C,\emptyset)$
apply (*rule sorted-map-cmono*[*OF s*])
using *mp*
by (*auto simp*: *tvars-pat-def intro*!: *restrict-map-mono-right*)
from *match-complete-wrt-o-imp*[*OF this m*] *mp*
show *?thesis* **by** *auto*
qed

lemma *pat-completeI*:
assumes *r*: $\forall\,\sigma :_s \mathcal{V}\ |`$ *tvars-pat pp* $\rightarrow \mathcal{T}(C,\emptyset::'v \rightharpoonup 's).\ \exists$ *mp* $\in$ *pp. match-complete-wrt* σ *mp*
shows *pat-complete C pp*
proof (*unfold pat-complete-def*, *safe*)
fix σ **assume** *s*: $\sigma :_s \mathcal{V}\ |`$ *tvars-pat pp* $\rightarrow \mathcal{T}(C)$
then have $\sigma \circ_s undefined :_s \mathcal{V}\ |`$ *tvars-pat pp* $\rightarrow \mathcal{T}(C,\emptyset)$
by (*simp add*: *subst-compose-sorted-map*)
from *r*[*rule-format*, *OF this*]
obtain *mp* **where** *mp*: *mp* $\in$ *pp* **and** *m*: *match-complete-wrt* ($\sigma \circ_s$ *undefined*::-$\Rightarrow$(-,$'v$) *term*) *mp*
by *auto*
have $\sigma :_s \mathcal{V}\ |`$ *tvars-match mp* $\rightarrow \mathcal{T}(C)$
apply (*rule sorted-map-cmono*[*OF s restrict-map-mono-right*])
using *mp* **by** (*auto simp*: *tvars-pat-def*)
from *match-complete-wrt-o-imp*[*OF this m*] *mp*
show *Bex pp* (*match-complete-wrt* σ) **by** *auto*

qed

lemma *tvars-pat-empty*[*simp*]: *tvars-pat* {} = {}
by (*simp add*: *tvars-pat-def*)

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

abbreviation *pats-complete* :: ('*f*,'*s*) *ssig* ⇒ ('*f*,'*v*,'*s*)*pats-problem-set* ⇒ *bool* **where**
pats-complete C P ≡ ∀ *pp* ∈ *P*. *pat-complete C pp*

6.2 Definition of Algorithm – Inference Rules

A function to compute for a variable x all substitution that instantiate x by $c(x_n, ..., x_{n+a})$ where c is a constructor of arity a and n is a parameter that determines from where to start the numbering of variables.

definition τ*c* :: *nat* ⇒ *nat* × '*s* ⇒ '*f* × '*s list* ⇒ ('*f*,*nat* × '*s*)*subst* **where**
τ*c 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*×'*s*)*term* ⇒ ('*f*,'*v*×'*s*)*term* ⇒ ('*v*×'*s*) *list option* **where**
conflicts (*Var x*) (*Var y*) = (*if x* = *y then Some* [] *else*
if snd x = *snd y then Some* [*x*,*y*] *else None*)
| *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*

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:
 shows Conflict-Clash s t ⟹
  ∃ p. p ∈ poss s ∧ p ∈ poss t ∧
  (is-Fun (s |-p) ∧ is-Fun (t |-p) ∧ root (s |-p) ≠ root (t |- p) ∨
  (∃ x y. s |-p = Var x ∧ t |-p = Var y ∧ snd x ≠ snd y))
  (is ?B1 ⟹ ?B2)
  and Conflict-Var s t x ⟹
  ∃ p . p ∈ poss s ∧ p ∈ poss t ∧ s |-p ≠ t |-p ∧
  (s |-p = Var x ∨ t |-p = Var x)
  (is ?C1 x ⟹ ?C2 x)
  and s ≠ t ⟹ ∃ x. Conflict-Clash s t ∨ Conflict-Var s t x
  and Conflict-Var s t x ⟹ x ∈ vars s ∪ vars t
  and conflicts s t = Some [] ⟷ s = t (is ?A)
proof −
 let ?B = ?B1 ⟶ ?B2
 let ?C = λ x. ?C1 x ⟶ ?C2 x
 {
  fix x
  have (conflicts s t = Some [] ⟶ s = t) ∧ ?B ∧ ?C x
  proof (induction s arbitrary: t)
   case (Var y t)
   thus ?case by (cases t, cases y, 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 ∈ set ss by auto
          from Fun.IH[OF this, of ts ! i] confl obtain p
            where p: p ∈ poss (ss ! i) ∧ p ∈ poss (ts ! i) ∧
(is-Fun (ss ! i |- p) ∧ is-Fun (ts ! i |- p) ∧ root (ss ! i |- p) ≠ root (ts ! i |- p) ∨
(∃ x y. ss!i |- p = Var x ∧ ts!i |- p = Var y ∧ snd x ≠ snd y))
            by force
          from p have p: ∃ p. p ∈ poss ?s ∧ p ∈ poss t ∧
(is-Fun (?s |- p) ∧ is-Fun (t |- p) ∧ root (?s |- p) ≠ root (t |- p) ∨
(∃ x y. ?s |- p = Var x ∧ t |- p = Var y ∧ snd x ≠ snd y))
            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 ⟹ 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 ∈ set (the (conflicts ?s t))
            hence x ∈ set (concat xss) unfolding res by auto
            then obtain xs where xs: xs ∈ set xss and x: x ∈ 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 ∈ set ss by auto
            from Fun.IH[OF this, of ts ! i, unfolded rec[OF i, folded xs]] x
            obtain p where p ∈ poss (ss ! i) ∧ p ∈ poss (ts ! i) ∧ ss ! i |- p ≠ ts
! i |- p ∧ (ss ! i |- p = Var x ∨ ts ! i |- p = Var x)
              by auto
            hence ∃ p. p ∈ poss ?s ∧ p ∈ poss t ∧ ?s |- p ≠ t |- p ∧ (?s |- p =
Var x ∨ t |- 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 ∈ 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 ⟹ ?B2 and C: ?C1 x ⟹ ?C2 x by blast+
show ?A
proof
 assume s = t
 with B have conflicts s t ≠ None by force
 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 ≠ t
 show ∃ x. Conflict-Clash s t ∨ Conflict-Var s t x
 proof (cases conflicts s t)
  case (Some xs)
  with ‹?A› diff obtain x where x ∈ set xs by (cases xs, auto)
  thus ?thesis unfolding Some
   apply auto
   by (metis surj-pair)
 qed auto
}
assume Conflict-Var s t x
with C obtain p where p ∈ poss s p ∈ poss t (s |- p = Var x ∨ t |- p = Var
x)
 by blast
thus x ∈ vars s ∪ 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*

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 *cd-sort* :: *'s* ⇒ *nat* — a function to compute finite cardinality of a sort
and *improved* :: *bool* — if improved = False, then FSCD-version of algorithm is used; if improved = True, the better journal version (under development) is used.
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 *lvars-disj-mp* :: *'v list* ⇒ (*'f*,*'v*,*'s*)*match-problem-set* ⇒ *bool* **where**
lvars-disj-mp ys mp = (⋃ (*vars* ‘ *snd* ‘ *mp*) ∩ *set ys* = {} ∧ *distinct ys*)

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 *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 mp = *subst-left τ* ‘ *mp*

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 pp = *subst-match-problem-set τ* ‘ *pp*

definition *τs* :: *nat* ⇒ *nat* × *'s* ⇒ (*'f*,*nat* × *'s*)*subst set* **where**
τs n x = {*τc n x* (*f*,*ss*) | *f ss*. *f* : *ss* → *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-problem-set* ⇒ (*'f*,*'v*,*'s*)*match-problem-set* ⇒ *bool*
(**infix** ‹→$_s$› *50*) **where**

mp-decompose: $length\ ts = length\ ls \Longrightarrow insert\ (Fun\ f\ ts,\ Fun\ f\ ls)\ mp \rightarrow_s set\ (zip\ ts\ ls) \cup mp$
| *mp-match*: $x \notin \bigcup\ (vars\ `\ snd\ `\ mp) \Longrightarrow insert\ (t,\ Var\ x)\ mp \rightarrow_s mp$
| *mp-identity*: $mp \rightarrow_s mp$
| *mp-decompose'*: $mp \cup mp' \rightarrow_s (\bigcup\ (t,\ l) \in mp.\ set\ (zip\ (args\ t)\ (map\ Var\ ys))) \cup mp'$
if $\bigwedge t\ l.\ (t,l) \in mp \Longrightarrow l = Var\ y \wedge root\ t = Some\ (f,n)$
$\bigwedge t\ l.\ (t,l) \in mp' \Longrightarrow y \notin vars\ l$
lvars-disj-mp ys $(mp \cup mp')$ *length ys* $= n$
improved

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

inductive *pp-step* :: $('f,'v,'s)$*pat-problem-set* $\Rightarrow$ $('f,'v,'s)$*pat-problem-set* $\Rightarrow$ *bool*
(**infix** $\langle\Rightarrow_s\rangle$ *50*) **where**
pp-simp-mp: $mp \rightarrow_s mp' \Longrightarrow insert\ mp\ pp \Rightarrow_s insert\ mp'\ pp$
| *pp-remove-mp*: *mp-fail mp* $\Longrightarrow insert\ mp\ pp \Rightarrow_s pp$
| *pp-inf-var-conflict*: $pp \cup pp' \Rightarrow_s pp'$
if *Ball pp inf-var-conflict*
finite pp
Ball $(tvars\text{-}pat\ pp')\ (\lambda\ x.\ \neg\ inf\text{-}sort\ (snd\ x))$
$\neg$ *improved* $\Longrightarrow pp' = \{\}$

Note that in *pp-inf-var-conflict* 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.

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

inductive *P-step-set* :: $('f,'v,'s)$*pats-problem-set* $\Rightarrow$ $('f,'v,'s)$*pats-problem-set* $\Rightarrow$ *bool*
(**infix** $\langle\Rrightarrow_s\rangle$ *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-success pp* $\Longrightarrow insert\ pp\ P \Rrightarrow_s P$
| *P-instantiate*: *tvars-disj-pp* $\{n\ ..<\ n{+}m\}\ pp \Longrightarrow x \in tvars\text{-}pat\ pp \Longrightarrow$
$insert\ pp\ P \Rrightarrow_s \{subst\text{-}pat\text{-}problem\text{-}set\ \tau\ pp\ |.\ \tau \in \tau s\ n\ x\} \cup P$

6.3 Soundness of the inference rules

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-problem-set* ⇒ *bool* **where**
wf-match mp = (*snd* ‘ *tvars-match mp* ⊆ *S*)

lemma *wf-match-iff*: *wf-match mp* ⟷ (∀ (*x*,*ι*) ∈ *tvars-match mp*. *ι* ∈ *S*)
by (*auto simp*: *wf-match-def*)

lemma *tvars-match-subst*: *tvars-match* (*subst-match-problem-set σ mp*) = (⋃ (*t*,*l*) ∈ *mp*. *vars* (*t*·*σ*))
by (*auto simp*: *tvars-match-def subst-match-problem-set-def subst-left-def*)

lemma *wf-match-subst*:
assumes *s*: *σ* :$_s$ *V* |‘ *tvars-match mp* → *T*(*C'*,{*x* : *ι* *in* *V*. *ι* ∈ *S*})
shows *wf-match* (*subst-match-problem-set σ mp*)
apply (*unfold wf-match-iff tvars-match-subst*)
proof (*safe*)
fix *t l x ι* **assume** *tl*: (*t*,*l*) ∈ *mp* **and** *xt*: (*x*,*ι*) ∈ *vars* (*t*·*σ*)
from *xt* **obtain** *y κ* **where** *y*: (*y*,*κ*) ∈ *vars t* **and** *xy*: (*x*,*ι*) ∈ *vars* (*σ* (*y*,*κ*)) **by** (*auto simp*: *vars-term-subst*)
from *tl y* **have** (*y*,*κ*) : *κ* *in* *V* |‘ *tvars-match mp* **by** (*auto simp*: *hastype-restrict tvars-match-def*)
from *sorted-mapD*[*OF s this*]
have *σ* (*y*,*κ*) : *κ* *in* *T*(*C'*,{*x* : *ι* *in* *V*. *ι* ∈ *S*}).
from *hastype-in-Term-imp-vars*[*OF this xy*]
have (*x*,*ι*) : *ι* *in* {*x* : *ι* *in* *V*. *ι* ∈ *S*} **by** (*auto elim*!: *in-dom-hastypeE*)
then show *ι* ∈ *S* **by** *auto*
qed

definition *wf-pat* :: (*'f*,*'v*,*'s*)*pat-problem-set* ⇒ *bool* **where**
wf-pat pp = (∀ *mp* ∈ *pp*. *wf-match mp*)

lemma *wf-pat-subst*:
assumes *s*: *σ* :$_s$ *V* |‘ *tvars-pat pp* → *T*(*C'*,{*x* : *ι* *in* *V*. *ι* ∈ *S*})
shows *wf-pat* (*subst-pat-problem-set σ pp*)
apply (*unfold wf-pat-def subst-pat-problem-set-def*)
proof *safe*
fix *mp* **assume** *mp*: *mp* ∈ *pp*
show *wf-match* (*subst-match-problem-set σ mp*)
apply (*rule wf-match-subst*)
apply (*rule sorted-map-cmono*[*OF s*])
apply (*rule restrict-map-mono-right*) **using** *mp* **by** (*auto simp*: *tvars-pat-def*)
qed

definition *wf-pats* :: (*'f*,*'v*,*'s*)*pats-problem-set* ⇒ *bool* **where**
wf-pats P = (∀ *pp* ∈ *P*. *wf-pat pp*)

lemma *wf-pat-iff*: *wf-pat pp* $\longleftrightarrow$ $(\forall (x,\iota) \in$ *tvars-pat pp*. $\iota \in S)$
by (*auto simp*: *wf-pat-def tvars-pat-def wf-match-iff*)

The reduction of match problems preserves completeness.

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*
next
case $*$: (*mp-decompose' mp y f n mp' ys*)
note $* = *$[*unfolded lvars-disj-mp-def*]
let *?mpi* = $(\bigcup (t,\ l) \in mp.$ *set* (*zip* (*args t*) (*map Var ys*)))
let *?y* = *Var y*
show *?case*
proof
assume *match-complete-wrt* σ (*?mpi* $\cup$ *mp'*)
from *this*[*unfolded match-complete-wrt-def*] **obtain** μ
where *match*: $\bigwedge$ *t l*. $(t,l) \in$ *?mpi* $\Longrightarrow t \cdot \sigma = l \cdot \mu$
and *match'*: $\bigwedge$ *t l*. $(t,l) \in mp' \Longrightarrow t \cdot \sigma = l \cdot \mu$ **by** *force*
let $?\mu = \mu(\ y := Fun\ f\ (map\ \mu\ ys))$
show *match-complete-wrt* σ (*mp* $\cup$ *mp'*) **unfolding** *match-complete-wrt-def*
proof (*intro exI*[*of* - *?*μ] *ballI*, *elim UnE*; *clarify*)
fix *t l*
{
assume $(t,l) \in mp'$
from *match'*[*OF this*] $*(2)$[*OF this*]
show $t \cdot \sigma = l \cdot ?\mu$ **by** (*auto intro*: *term-subst-eq*)
}
assume *tl*: $(t,l) \in mp$
from $*(1)$[*OF this*] **obtain** *ts* **where** *l*: $l = Var\ y$ **and** *t*: $t = Fun\ f\ ts$
and *lts*: *length ts* = *n* **by** (*cases t*, *auto*)
{
fix *ti yi*

```
      assume (ti,yi) ∈ set (zip ts ys)
      hence (ti, Var yi) ∈ set (zip (args t) (map Var ys))
        using t lts ‹length ys = n› by (force simp: set-conv-nth)
      hence (ti, Var yi) ∈ ?mpi using tl by blast
      from match[OF this] have μ yi = ti · σ by simp
    } note yi = this
    show t · σ = l · ?μ unfolding l t using yi lts ‹length ys = n›
      by (force intro!: nth-equalityI simp: set-zip)
  qed
 next
  assume match-complete-wrt σ (mp ∪ mp')
  from this[unfolded match-complete-wrt-def]
  obtain μ where match: ⋀ t l. (t,l) ∈ mp ⟹ t · σ = l · μ
    and match': ⋀ t l. (t,l) ∈ mp' ⟹ t · σ = l · μ by force
  define μ' where μ' = (λ x. case map-of (zip ys (args (μ y))) x of
    None ⇒ μ x | Some Ti ⇒ Ti)
  show match-complete-wrt σ (?mpi ∪ mp')
    unfolding match-complete-wrt-def
  proof (intro exI[of - μ'] ballI, elim UnE; clarify)
    fix t l
    assume tl: (t,l) ∈ mp'
    from ∗(3) tl have vars: vars l ∩ set ys = {} by force
    hence map-of (zip ys (args (μ y))) x = None if x ∈ vars l for x
     using that by (meson disjoint-iff map-of-SomeD option.exhaust set-zip-leftD)
    with match'[OF tl]
    show t · σ = l · μ' by (auto intro!: term-subst-eq simp: μ'-def)
  next
    fix t l ti and vyi :: ('f,-)term
    assume tl: (t,l) ∈ mp
      and i: (ti,vyi) ∈ set (zip (args t) (map Var ys))
    from ∗(1)[OF tl] obtain ts where l: l = Var y and t: t = Fun f ts
      and lts: length ts = n by (cases t, auto)
    from i lts obtain i where i: i < n and ti: ti = ts ! i and yi: vyi = Var (ys
! i)
      unfolding set-zip using ‹length ys = n› t by auto
    from match[OF tl] have mu-y: μ y = Fun f ts · σ unfolding l t by auto
    have yi: vyi · μ' = args (μ y) ! i unfolding μ'-def yi
      using i lts ‹length ys = n› ∗(3) mu-y
      by (force split: option.splits simp: set-zip distinct-conv-nth)
    also have ... = ti · σ unfolding mu-y ti using i lts by auto
    finally show ti · σ = vyi · μ' ..
  qed
 qed
qed auto

lemma mp-fail-pcorrect1:
 assumes mp-fail mp σ :ₛ sort-annotated |` tvars-match mp → 𝒯(C,X)
 shows ¬ match-complete-wrt σ mp
 using assms
```

```
proof (induct mp rule: mp-fail.induct)
  case ∗: (mp-clash f ts g ls mp)
  {
    assume length ts ≠ length ls
     hence (map (λt. t · μ) ls = map (λt. t · σ) ts) = False for σ :: ('f,nat ×
's,'a)gsubst and μ
      by (metis length-map)
  } note len = this
  from ∗ show ?case unfolding match-complete-wrt-def
    apply (auto simp: len split: prod.splits)
    using map-eq-imp-length-eq by force
next
  case ∗: (mp-clash' s t x mp)
  from conflicts(1)[OF ∗(1)]
  obtain po where po: po ∈ poss s po ∈ poss t
    and disj: is-Fun (s |- po) ∧ is-Fun (t |- po) ∧ root (s |- po) ≠ root (t |- po) ∨
    (∃ x y. s |- po = Var x ∧ t |- po = Var y ∧ snd x ≠ snd y)
    by auto
  show ?case
  proof
    assume match-complete-wrt σ ({(s, Var x), (t, Var x)} ∪ mp)
    from this[unfolded match-complete-wrt-def]
    have eq: s · σ |-po = t · σ |-po by auto
    from disj
    show False
    proof (elim disjE conjE exE)
      assume ∗: is-Fun (s |- po) is-Fun (t |- po) root (s |- po) ≠ root (t |- po)
      from eq have root (s · σ |-po) = root (t · σ |-po) by auto
      also have root (s · σ |-po) = root (s |-po · σ) using po by auto
      also have ... = root (s |-po) using ∗ by (cases s |- po, auto)
       also have root (t · σ |-po) = root (t |-po · σ) using po by (cases t |- po,
auto)
      also have ... = root (t |-po) using ∗ by (cases t |- po, auto)
      finally show False using ∗ by auto
    next
      fix y z assume y: s |- po = Var y and z: t |- po = Var z and ty: snd y ≠
snd z
      from y z eq po have yz: σ y = σ z by auto
      have y ∈ vars-term s z ∈ vars-term t
        using po[THEN vars-term-subt-at] y z by auto
      then
      have σ y : snd y in 𝒯(C,X) σ z : snd z in 𝒯(C,X)
      by (auto intro!: ∗(2)[THEN sorted-mapD] simp: hastype-restrict tvars-match-def)
      with ty yz show False by (auto simp: has-same-type)
    qed
  qed
next
  case ∗: (mp-clash-sort s t x mp)
  show ?case
```

proof
assume *match-complete-wrt* σ $(\{(s,\ Var\ x),\ (t,\ Var\ x)\} \cup mp)$
from *this*[*unfolded match-complete-wrt-def*]
have *eq*: $s \cdot \sigma = t \cdot \sigma$ **by** *auto*
define V **where** $V = tvars\text{-}match\ (\{(s,\ Var\ x),\ (t,\ Var\ x)\} \cup mp)$
from $*(2)$ **have** σ: $\sigma :_s \mathcal{V}$ |‘ $V \to \mathcal{T}(C,X)$ **unfolding** *V-def* **.**
have *vars*: *vars* $s \cup$ *vars* $t \subseteq V$ **unfolding** *V-def tvars-match-def* **by** *auto*
show *False*
proof (*cases None* $\in \{\mathcal{T}(C,\mathcal{V})\ s,\ \mathcal{T}(C,\mathcal{V})\ t\}$)
case *False*
from *False* **obtain** $\sigma s\ \sigma t$ **where** *st*: $s : \sigma s$ *in* $\mathcal{T}(C,\mathcal{V})\ t : \sigma t$ *in* $\mathcal{T}(C,\mathcal{V})$
by (*cases* $\mathcal{T}(C,\mathcal{V})$ *s*; *cases* $\mathcal{T}(C,\mathcal{V})$ *t*; *auto simp*: *hastype-def*)
from *st*(*1*) *vars* σ **have** $(s \cdot \sigma) : \sigma s$ *in* $\mathcal{T}(C,X)$
by (*meson le-supE restrict-map-mono-right sorted-algebra.eval-has-same-type-vars sorted-map-cmono*
term.sorted-algebra-axioms)
moreover from *st*(*2*) *vars* σ **have** $(t \cdot \sigma) : \sigma t$ *in* $\mathcal{T}(C,X)$
by (*meson le-supE restrict-map-mono-right sorted-algebra.eval-has-same-type-vars sorted-map-cmono*
term.sorted-algebra-axioms)
ultimately have $\sigma s = \sigma t$ **unfolding** *eq hastype-def* **by** *auto*
with *st* $*(1)$ **show** *False* **by** (*auto simp*: *hastype-def*)
next
case *True*
have $\exists\ s\ \sigma s.$ *vars* $s \subseteq V \wedge s \cdot \sigma : \sigma s$ *in* $\mathcal{T}(C,X) \wedge \mathcal{T}(C,\mathcal{V})\ s =$ *None*
proof (*cases* $\mathcal{T}(C,\mathcal{V})$ *s*)
case *None*
with $*(1)$ **obtain** σt **where** $t : \sigma t$ *in* $\mathcal{T}(C,\mathcal{V})$ **by** (*cases* $\mathcal{T}(C,\mathcal{V})$ *t*; *force simp*: *hastype-def*)
from *this vars* σ **have** $(t \cdot \sigma) : \sigma t$ *in* $\mathcal{T}(C,X)$
by (*meson le-supE restrict-map-mono-right sorted-algebra.eval-has-same-type-vars sorted-map-cmono*
term.sorted-algebra-axioms)
from *this*[*folded eq*] *None vars* **show** *?thesis* **by** *auto*
next
case (*Some* σs)
with *True* **have** *None*: $\mathcal{T}(C,\mathcal{V})\ t =$ *None* **and** *Some*: $s : \sigma s$ *in* $\mathcal{T}(C,\mathcal{V})$ **by** (*auto simp*: *hastype-def*)
from *Some vars* σ **have** $(s \cdot \sigma) : \sigma s$ *in* $\mathcal{T}(C,X)$
by (*meson le-supE restrict-map-mono-right sorted-algebra.eval-has-same-type-vars sorted-map-cmono*
term.sorted-algebra-axioms)
from *this*[*unfolded eq*] *None vars* **show** *?thesis* **by** *auto*
qed
then obtain $s\ \sigma s$ **where** *vars* $s \subseteq V\ s \cdot \sigma$: σs *in* $\mathcal{T}(C,X)\ \mathcal{T}(C,\mathcal{V})\ s =$ *None* **by** *auto*
thus *False*
proof (*induct s arbitrary*: σs)
case (*Fun f ss* τ)

```
      hence mem: Fun f (map (λs. s · σ) ss) : τ in 𝒯(C,X) by auto
      from this[unfolded Fun-hastype]
      obtain τs where f: f : τs → τ in C and args: map (λs. s · σ) ss :ₗ τs in
𝒯(C,X) by auto
      {
        fix s
        assume s ∈ set ss
        hence s · σ ∈ set (map (λs. s · σ) ss) by auto
        hence ∃ τ. s · σ : τ in 𝒯(C,X)
       by (metis Fun-in-dom-imp-arg-in-dom mem hastype-imp-dom in-dom-hastypeE)
      } note arg = this
      show ?case
      proof (cases ∃ s ∈ set ss. 𝒯(C,𝒱) s = None)
        case True
       then obtain s where s: s ∈ set ss and None: 𝒯(C,𝒱) s = None by auto
        from arg[OF s] obtain τ where Some: s · σ : τ in 𝒯(C,X)  by auto
        from Fun(1)[OF s - Some None] s Fun(2) show False by auto
      next
        case False
        have Fun f ss : τ in 𝒯(C,𝒱)
         proof (intro Fun-hastypeI[OF f], unfold list-all2-conv-all-nth, intro conjI
allI impI)
          show length ss = length τs using args[unfolded list-all2-conv-all-nth] by
auto
          fix i
          assume i: i < length ss
          hence ssi: ss ! i ∈ set ss by auto
          with False obtain τi where type: ss ! i : τi in 𝒯(C,𝒱) by (auto simp:
hastype-def)
          from ssi Fun(2) have vars: vars (ss ! i) ⊆ V by auto
          from vars type σ have ss ! i · σ : τi in 𝒯(C,X)
        by (meson restrict-map-mono-right sorted-map-cmono term.eval-has-same-type-vars)
          moreover from args i have ss ! i · σ : τs ! i in 𝒯(C,X)
            unfolding list-all2-conv-all-nth by auto
          ultimately have τi = τs ! i by (auto simp: hastype-def)
          with type show ss ! i : τs ! i in 𝒯(C,𝒱) by auto
        qed
         with Fun(4) show False unfolding hastype-def using not-None-eq by
blast
      qed
    qed auto
  qed
 qed
qed

lemma mp-fail-pcorrect:
  assumes f: mp-fail mp and s: σ :ₛ {x : ι in 𝒱. ι ∈ S} → 𝒯(C) and wf: wf-match
mp
  shows ¬ match-complete-wrt σ mp
```

apply (*rule mp-fail-pcorrect1*[*OF f*])
apply (*rule sorted-map-cmono*[*OF s*])
using *wf* **by** (*auto intro*!: *subssetI simp*: *hastype-restrict wf-match-iff*)

end

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 cd-sort*
for *S* **and** *C* :: ('*f*,'*s*)*ssig*
and *m Cl inf-sort cd-sort* +
assumes *not-empty-sort*: $\bigwedge$ *s*. *s* ∈ *S* ⟹ ¬ *empty-sort C s*
and *C-sub-S*: $\bigwedge$ *f ss s*. *f* : *ss* → *s in C* ⟹ *insert s* (*set ss*) ⊆ *S*
and *m*: $\bigwedge$ *f ss s*. *f* : *ss* → *s in C* ⟹ *length ss* ≤ *m*
and *finite-C*: *finite* (*dom C*)
and *inf-sort*: $\bigwedge$*s*. *s* ∈ *S* ⟹ *inf-sort s* ⟷ ¬ *finite-sort C s*
and *Cl*: $\bigwedge$ *s*. *set* (*Cl s*) = {(*f*,*ss*). *f* : *ss* → *s in C*}
and *Cl-len*: $\bigwedge$ σ. *Ball* (*length* ‘ *snd* ‘ *set* (*Cl* σ)) (λ *a*. *a* ≤ *m*)
and *cd*: $\bigwedge$*s*. *s* ∈ *S* ⟹ *cd-sort s* = *card-of-sort C s*
begin

lemma *sorts-non-empty*: *s* ∈ *S* ⟹ ∃ *t*. *t* : *s in* $\mathcal{T}$(*C*,∅)
apply (*drule not-empty-sort*)
by (*auto elim*: *not-empty-sortE*)

lemma *inf-sort-not-bdd*: *s* ∈ *S* ⟹ ¬ *bdd-above* (*size* ‘ {*t* . *t* : *s in* $\mathcal{T}$(*C*,∅)}) ⟷ *inf-sort s*
apply (*subst finite-sig-bdd-above-iff-finite*[*OF finite-C*])
by (*auto simp*: *inf-sort finite-sort*)

lemma *C-nth-S*: *f* : *ss* → *s in C* ⟹ *i* < *length ss* ⟹ *ss*!*i* ∈ *S*
using *C-sub-S* **by** *force*

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

Preservation of well-formedness

lemma *mp-step-wf*: *mp* →$_s$ *mp*′ ⟹ *wf-match mp* ⟹ *wf-match mp*′
unfolding *wf-match-def tvars-match-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*)
next
case ∗: (*mp-decompose*′ *mp y f n mp*′ *ys*)

```
  from *(1) *(6)
  show ?case
    apply (auto dest!: set-zip-leftD)
    subgoal for - - t by (cases t; force)
    subgoal for - - t by (cases t; force)
    done
qed auto

lemma pp-step-wf: pp ⇒ₛ pp' ⟹ wf-pat pp ⟹ wf-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 ⇛ₛ P' ⟹ wf-pats P ⟹ wf-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 ∈ S and p: wf-pat p unfolding wf-pat-def wf-match-def
tvars-pat-def by auto
  {
    fix τ
    assume tau: τ ∈ τs n x
    from tau[unfolded τs-def τc-def, simplified]
    obtain f sorts where f: f : sorts → 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
    from C-sub-S[OF f] have sS: ?s ∈ S and xs: snd ‘ set ?xs ⊆ S
      unfolding set-conv-nth set-zip by auto
    {
      fix mp y
      assume mp: mp ∈ p  and y ∈ tvars-match (subst-left τ ‘ mp)
      then obtain s t where y: y ∈ vars (s · τ) and st: (s,t) ∈ mp
        unfolding tvars-match-def subst-left-def by auto
      from y have y ∈ vars s ∪ set ?xs unfolding vars-term-subst τ
        by (auto simp: subst-def split: if-splits)
      hence snd y ∈ snd ‘ vars s ∪ snd ‘ set ?xs by auto
      also have ... ⊆ snd ‘ vars s ∪ S using xs by auto
      also have ... ⊆ S using p mp st
        unfolding wf-pat-def wf-match-def tvars-match-def by force
      finally have snd y ∈ 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

definition $\sigma g :: nat \times 's \Rightarrow ('f,'v)\ term$ **where**
$\sigma g\ x = (SOME\ t.\ t : snd\ x\ in\ \mathcal{T}(C,\emptyset))$

lemma *σg*: $\sigma g :_s \{x : \iota\ in\ sort\text{-}annotated.\ \iota \in S\} \to \mathcal{T}(C,\emptyset)$
using *sorts-non-empty*[*THEN someI-ex*]
by (*auto intro*!: *sorted-mapI simp*: *σg-def*)

lemma *wf-pat-complete-iff*:
assumes *wf-pat pp*
shows *pat-complete C pp* $\longleftrightarrow$ $(\forall \sigma :_s \{x : \iota\ in\ \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C).\ \exists\ mp \in pp.$ *match-complete-wrt σ mp*)
(**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *l*: *?l*
show *?r*
proof (*intro allI impI*)
fix $\sigma :: nat \times 's \Rightarrow$ -
assume *s*: $\sigma :_s \{x : \iota\ in\ \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C)$
have $\sigma :_s \mathcal{V}\ |`$ *tvars-pat pp* $\to \mathcal{T}(C)$
apply (*rule sorted-map-cmono*[*OF s*])
using *assms* **by** (*auto intro*!: *subssetI simp*: *hastype-restrict wf-pat-iff*)
from *pat-completeD*[*OF l this*] **show** ∃ *mp*∈*pp. match-complete-wrt σ mp*.
qed
next
assume *r*: *?r*
show *?l*
proof (*unfold pat-complete-def*, *safe*)
fix *σ* **assume** *s*: $\sigma :_s \mathcal{V}\ |`$ *tvars-pat pp* $\to \mathcal{T}(C)$
define σ' **where** $\sigma'\ x \equiv$ *if x* ∈ *tvars-pat pp then σ x else σg x* **for** *x*
have $\sigma' :_s \{x : \iota\ in\ \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C)$
by (*auto intro*!: *sorted-mapI sorted-mapD*[*OF s*] *sorted-mapD*[*OF σg*] *simp*: *σ′-def hastype-restrict*)
from *r*[*rule-format*, *OF this*]
obtain *mp* **where** *mp*: *mp* ∈ *pp* **and** *m*: *match-complete-wrt σ′ mp* **by** *auto*
have [*simp*]: *x* ∈ *tvars-match mp* ⟹ *σ x* = *σ′ x* **for** *x* **using** *mp* **by** (*auto simp*: *σ′-def tvars-pat-def*)
from *m* **have** *match-complete-wrt σ mp* **by** (*simp cong*: *match-complete-wrt-cong*)
with *mp* **show** *Bex pp* (*match-complete-wrt σ*) **by** *auto*
qed
qed

lemma *wf-pats-complete-iff*:
assumes *wf*: *wf-pats P*

shows *pats-complete C P* $\longleftrightarrow$
($\forall \sigma :_s \{x : \iota \ in \ \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C).\ \forall pp \in P.\ \exists mp \in pp.\ match\text{-}complete\text{-}wrt\ \sigma\ mp$)
(**is** *?l* $\longleftrightarrow$ *?r*)
proof *safe*
fix σ *pp* **assume** *s*: $\sigma :_s \{x : \iota \ in \ \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C)$ **and** *pp*: $pp \in P$
have *s2*: $\sigma :_s \mathcal{V}$ |‘ *tvars-pats P* $\to \mathcal{T}(C)$
apply (*rule sorted-map-cmono*[*OF s*])
using *wf*
by (*auto intro*!: *subssetI simp*: *hastype-restrict wf-pats-def wf-pat-iff tvars-pats-def split*: *prod.splits*)
assume *?l*
with *pp* **have** *comp*: *pat-complete C pp* **by** *auto*
from *wf pp* **have** *wf-pat pp* **by** (*auto simp*: *wf-pats-def*)
from *comp*[*unfolded wf-pat-complete-iff*[*OF this*], *rule-format*, *OF s*]
show $\exists mp \in pp.$ *match-complete-wrt* σ *mp*.
next
fix *pp* **assume** *pp*: $pp \in P$
assume *r*[*rule-format*]: *?r*
from *wf pp* **have** *wf-pat pp* **by** (*auto simp*: *wf-pats-def*)
note $* =$ *wf-pat-complete-iff*[*OF this*]
show *pat-complete C pp*
apply (*unfold* $*$) **using** *r*[*OF - pp*] **by** *auto*
qed

lemma *inf-var-conflictD*: **assumes** *inf-var-conflict mp*
shows $\exists\ p\ s\ t\ x\ y.$
$(s, Var\ x) \in mp \wedge (t, Var\ x) \in mp \wedge s\ |\text{-}p = Var\ y \wedge s\ |\text{-}\ p \neq t\ |\text{-}p \wedge$
$p \in poss\ s \wedge p \in poss\ t \wedge$ *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

lemmas *cg-term-vars* = *hastype-in-Term-empty-imp-vars*

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

lemma *pp-step-pcorrect*:
$pp \Rightarrow_s pp' \Longrightarrow$ *wf-pat pp* $\Longrightarrow$ *pat-complete C pp* = *pat-complete C pp*$'$
proof (*induct pp pp*$'$ *rule*: *pp-step.induct*)
case $*$: (*pp-simp-mp mp mp*$'$ *pp*)
with *mp-step-wf*[*OF* $*$(*1*)]
have *wf-pat* (*insert mp*$'$ *pp*) **by** (*auto simp*: *wf-pat-def*)
with $*$(*2*) *mp-step-pcorrect*[*OF* $*$(*1*)]
show *?case* **by** (*auto simp*: *wf-pat-complete-iff*)
next

```
  case ∗: (pp-remove-mp mp pp)
  from mp-fail-pcorrect[OF ∗(1)] ∗(2)
  show ?case by (auto simp: wf-pat-complete-iff wf-pat-def)
next
  case ∗: (pp-inf-var-conflict pp pp')
  note wf = ‹wf-pat (pp ∪ pp')› and fin = ‹finite pp›
  hence wf-pat pp and wfpp': wf-pat pp' by (auto simp: wf-pat-def)
  with wf have easy: pat-complete C pp' ⟹ pat-complete C (pp ∪ pp')
    by (auto simp: wf-pat-complete-iff)
  {
    assume pp: pat-complete C (pp ∪ pp')
    have pat-complete C pp' unfolding wf-pat-complete-iff[OF wfpp']
    proof (intro allI impI)
      fix δ
      assume δ: δ :s {x : ι in V. ι ∈ S} → T(C)
       define conv :: ('f,unit) term ⇒ ('f, nat × 's) term where conv t = t ·
undefined for t
       define conv' :: ('f, nat × 's) term ⇒ ('f, unit) term where conv' t = t ·
undefined for t
      define confl' :: ('f, nat × 's) term ⇒ ('f, nat × 's)term ⇒ nat × 's ⇒ bool
where confl' = (λ sp tp y.
          sp = Var y ∧ inf-sort (snd y) ∧ sp ≠ tp)
     define P1 where P1 = (λ mp s t x y p. mp ∈ pp ⟶ (s, Var x) ∈ mp ∧ (t,
Var x) ∈ mp ∧ p ∈ poss s ∧ p ∈ poss t ∧ confl' (s |- p) (t |- p) y)
     {
       fix mp
       assume mp ∈ pp
       hence inf-var-conflict mp using ∗ by auto
       from inf-var-conflictD[OF this]
       have ∃ s t x y p. P1 mp s t x y p unfolding P1-def confl'-def by force
     }
     hence ∀ mp. ∃ s t x y p. P1 mp s t x y p unfolding P1-def by blast
     from choice[OF this] obtain s where ∀ mp. ∃ t x y p. P1 mp (s mp) t x y
p by blast
     from choice[OF this] obtain t where ∀ mp. ∃ x y p. P1 mp (s mp) (t mp)
x y p by blast
    from choice[OF this] obtain x where ∀ mp. ∃ y p. P1 mp (s mp) (t mp) (x
mp) y p by blast
     from choice[OF this] obtain y where ∀ mp. ∃ p. P1 mp (s mp) (t mp) (x
mp) (y mp) p by blast
     from choice[OF this] obtain p where ∀ 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 ⇒ nat × 's ⇒ bool where
       var-ind i x = (x ∈ y ‘ pp ∧ index x ∈ {..<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*,∅))))
∧ (*snd x* ∈ *S* ⟶ ¬ *inf-sort* (*snd x*) ⟶ *σ x* = *conv* (*δ x*))) **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 x*
assume *var-ind i x*
from *this*[*unfolded var-ind-def*] **obtain** *i*
where *z*: *x* ∈ *y* ‘ *pp index x* = *i* **by** *blast*
from *z* **obtain** *mp* **where** *mp* ∈ *pp* **and** *index* (*y mp*) = *i* **and** *x* = *y mp* **by** *auto*
with *P1*[*OF this*(*1*), *unfolded confl'-def*] **have** *inf*: *inf-sort* (*snd x*) **by** *auto*
} note *var-ind-inf* = *this*
{
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 conv* (*δ x*) *else Fun undefined* []) **for** *x*
have *σ*: *cg-subst-ind 0 σ* **unfolding** *cg-subst-ind-def*
proof (*intro allI impI conjI*)
fix *x*
show *var-ind 0 x* ⟹ *σ x* = *Var x* **unfolding** *σ-def* **by** *auto*
show ¬ *var-ind 0 x* ⟹ *vars* (*σ x*) = {}
unfolding *σ-def conv-def* **using** *δ*[*THEN sorted-mapD*, *of x*]
by (*auto simp*: *vars-term-subst hastype-in-Term-empty-imp-vars*)
show ¬ *var-ind 0 x* ⟹ *snd x* ∈ *S* ⟹ *σ x* : *snd x in* $\mathcal{T}$(*C*,∅)
using *δ*[*THEN sorted-mapD*, *of x*]
unfolding *σ-def conv-def* **by** (*auto simp*: *σ-def intro*: *type-conversion*)
show *snd x* ∈ *S* ⟹ ¬ *inf-sort* (*snd x*) ⟹ *σ x* = *conv* (*δ x*)
unfolding *σ-def* **by** (*auto dest*: *var-ind-inf*)
qed
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: conv (δ
u) = Var z
                by (auto split: if-splits)
              from δ[THEN sorted-mapD, of u] uS contra
              have False by (cases δ u, auto simp: conv-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* ∈ *pp* **and** *index* (*y mp*) = *i* **and** *z* = *y mp* **by** *auto*
with *P1*[*OF this*(*1*), *unfolded confl'-def*] **have** *inf*: *inf-sort* (*snd z*)
and ∗: *p mp* ∈ *poss* (*s mp*) *s mp* |- *p mp* = *Var z* (*s mp*, *Var* (*x mp*)) ∈ *mp*
by *auto*
from ∗(*1*,*2*) **have** *z* ∈ *vars* (*s mp*) **using** *vars-term-subt-at* **by** *fastforce*
with ∗(*3*) **have** *z* ∈ *tvars-match mp* **unfolding** *tvars-match-def* **by** *force*
with ‹*mp* ∈ *pp*› *wf* **have** *snd z* ∈ *S* **unfolding** *wf-pat-def wf-match-def* **by** *auto*
from *not-bdd-above-natD*[*OF inf-sort-not-bdd*[*OF this*, *THEN iffD2*, *OF inf*]]
sorts-non-empty[*OF this*]
have ⋀ *n*. ∃ *t*. *t* : *snd z in* $\mathcal{T}$(*C*,∅::*nat*×'*s*⇀-) ∧ *n* < *size t* **by** *auto*
note *this inf*
} **note** *z-inf* = *this*

define *all-st* **where** *all-st* = (λ *mp*. *s mp* · σ) ‘ *pp* ∪ (λ *mp*. *t mp* · σ) ‘ *pp*
have *fin-all-st*: *finite all-st* **unfolding** *all-st-def* **using** ∗(*2*) **by** *simp*
define *d* :: *nat* **where** *d* = *Suc* (*Max* (*size* ‘ *all-st*))
from *z-inf*(*1*)[*of d*]
obtain *u* :: ('*f*,*nat*×'*s*) *term*
where *u*: *u* : *snd z in* $\mathcal{T}$(*C*,∅) **and** *du*: *d* ≤ *size u* **by** *auto*
have *vars-u*: *vars u* = {} **by** (*rule cg-term-vars*[*OF u*])
define σ' **where** σ' *x* = (*if x* = *z then u else* σ *x*) **for** *x*
have σ'*-def'*: σ' *x* = (*if x* ∈ *y* ‘ *pp* ∧ *index x* = *i then u else* σ *x*) **for** *x*
unfolding σ'*-def* **by** (*rule if-cong*, *insert bij z*, *auto simp*: *bij-betw-def inj-on-def*)
have *var-ind-conv*: *var-ind i x* = (*x* = *z* ∨ *var-ind* (*Suc i*) *x*) **for** *x*
proof
assume *x* = *z* ∨ *var-ind* (*Suc i*) *x*
thus *var-ind i x* **using** *z i* **unfolding** *var-ind-def* **by** *auto*
next
assume *var-ind i x*
hence *x*: *x* ∈ *y* ‘ *pp index x* ∈ {..<*n*} − {..<*i*} **unfolding** *var-ind-def* **by** *auto*
with *i* **have** *index x* = *i* ∨ *index x* ∈ {..<*n*} − {..<*Suc i*} **by** *auto*
thus *x* = *z* ∨ *var-ind* (*Suc i*) *x*
proof
assume *index x* = *i*
with *x*(*1*) *z bij* **have** *x* = *z* **by** (*auto simp*: *bij-betw-def inj-on-def*)
thus *?thesis* **by** *auto*
qed (*insert x*, *auto simp*: *var-ind-def*)
qed
have [*simp*]: *var-ind i z* **unfolding** *var-ind-conv* **by** *auto*
have [*simp*]: *var-ind* (*Suc i*) *z* = *False* **unfolding** *var-ind-def* **using** *z* **by**

```
auto
        have σz[simp]: σ z = Var z using σ[unfolded cg-subst-ind-def, rule-format,
of z] by auto
         have σ'-upd: σ' = σ(z := u) unfolding σ'-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,∅)
            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
```

```
        next
          fix x :: nat × 's
          assume ∗: snd x ∈ S ¬ inf-sort (snd x)
          with z-inf(2) have x ≠ z by auto
          hence σ' x = σ x unfolding σ'-def by auto
          thus σ' x = conv (δ x) using σ[unfolded cg-subst-ind-def, rule-format,
of x] ∗ by auto
        qed
        show ?case
        proof (intro exI[of - σ'] conjI σ' ballI)
          fix mp
          assume mp: mp ∈ pp
          define s' where s' = s mp · σ
          define t' where t' = t mp · σ
          from confl[rule-format, OF mp]
         obtain p where p: p ∈ poss s' p ∈ poss t' and confl: confl i (s' |- p) (t'
|- p) by (auto simp: s'-def t'-def)
          {
            fix s' t' :: ('f, nat × 's) term and p f ss x
            assume ∗: (s' |- p, t' |-p) = (Fun f ss, Var x) var-ind i x and p: p ∈
poss s' p ∈ poss t'
              and range-all-st: s' ∈ all-st
            hence s': s' · Var(z := u) |- p = Fun f ss · Var(z := u) (is - = ?s)
              and t': t' · 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 rangeσ: ∃ S. s' = S · σ by auto
            define s where s = ?s
            have ∃p. p ∈ poss (s' · Var(z := u)) ∧ p ∈ poss (t' · Var(z := u)) ∧
confl (Suc i) (s' · Var(z := u) |- p) (t' · 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' · Var(z := u) |- p = u unfolding t' by auto
             have ∃ p'. p' ∈ poss u ∧ p' ∈ poss s ∧ confl (Suc i) (s |- p') (u |- p')
              proof (cases ∃ x. x ∈ vars s ∧ var-ind (Suc i) x)
                case True
                then obtain x where xs: x ∈ vars s and x: var-ind (Suc i) x by
auto
                 from xs obtain p' where p': p' ∈ 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 ∈ 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 ∈ 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' ∈ poss (us ! j) ∧ p' ∈ poss (ss ! j) ∧ 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 rangeσ obtain S where sS: s' = S · σ by auto
    from p have vars (s' |- p) ⊆ vars s' by (metis vars-term-subt-at)
    also have ... = (⋃y∈vars S. vars (σ y)) unfolding sS by (simp
add: vars-term-subst)
    also have ... ⊆ (⋃y∈vars S. Collect (var-ind i))
    proof −
      {
        fix x y
        assume x ∈ 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) ⊆ Collect (var-ind i) by blast
  have vars s = vars (s' |- p · Var(z := u)) unfolding s-def s' fss by
auto
    also have ... = ⋃ (vars ‘ Var(z := u) ‘ vars (s' |- p)) by (simp
add: vars-term-subst)
  also have ... ⊆ ⋃ (vars ‘ Var(z := u) ‘ Collect (var-ind i)) using
sub by auto
    also have ... ⊆ 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 · Var(z := u) = u by auto
  have False
  proof (cases z ∈ vars (s' |- p))
    case True
    have diff: s' |- p ≠ Var z using ∗ by auto
    from True obtain C where id: s' |- p = C ⟨ Var z ⟩
      by (metis ctxt-supt-id vars-term-poss-subt-at)
    with diff have diff: C ≠ Hole by (cases C, auto)
    from eq[unfolded id, simplified] diff
    obtain C where C⟨u⟩ = u and C ≠ 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' ∈ size ‘ all-st using range-all-st by auto
    from False have s' |- p · Var(z := u) = s' |- p · 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 ... ≤ size s' using p(1)
      by (rule subt-size)
    also have ... ≤ Max (size ‘ all-st)
      using size fin-all-st by simp
    also have ... < d unfolding d-def by simp
    also have ... ≤ size u using du .
    finally show False by simp
  qed
}
hence s ≠ 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 ∃ j < length us. ss ! j ≠ us ! j
      by (auto simp: list-eq-nth-eq)
     then obtain j where j: j < length us and diff: ss ! j ≠ us ! j
```

by *auto*
from *j True* **have** *mem*: *ss* ! *j* ∈ *set ss us* ! *j* ∈ *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′ ∈ *poss* (*us* ! *j*) ∧ *p*′ ∈ *poss* (*ss* ! *j*) ∧ *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*′ ∈ *poss u p*′ ∈ *poss s* **and** *confl*: *confl* (*Suc i*) (*s* |- *p*′) (*u* |- *p*′) **by** *auto*
have *s*′′: *s*′ · *Var*(*z* := *u*) |- (*p* @ *p*′) = *s* |- *p*′ **unfolding** *s-def s*′[*symmetric*] **using** *p p*′ **by** *auto*
have *t*′′: *t*′ · *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* ∈ *poss* (*s*′ · *Var*(*z* := *u*)) **using** *p* **by** *auto*
moreover have *p*′ ∈ *poss* ((*s*′ · *Var*(*z* := *u*)) |- *p*) **using** *s*′ *p*′ *p* **unfolding** *s-def* **by** *auto*
ultimately show *p* @ *p*′ ∈ *poss* (*s*′ · *Var*(*z* := *u*)) **by** *simp*
have *p* ∈ *poss* (*t*′ · *Var*(*z* := *u*)) **using** *p* **by** *auto*
moreover have *p*′ ∈ *poss* ((*t*′ · *Var*(*z* := *u*)) |- *p*) **using** *t*′ *p*′ *p* **by** *auto*
ultimately show *p* @ *p*′ ∈ *poss* (*t*′ · *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*) ≠ (*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* ≠ *x*′ *var-ind i x var-ind i x*′
using *confl* **by** (*auto simp*: *confl-def split*: *term.splits*)
hence ∃ *p*. *p* ∈ *poss* (*s*′ · *Var*(*z* := *u*)) ∧ *p* ∈ *poss* (*t*′ · *Var*(*z* := *u*)) ∧ *confl* (*Suc i*) (*s*′ · *Var*(*z* := *u*) |- *p*) (*t*′ · *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*′ ∈ *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*′ ∈ *all-st* **unfolding** *t*′*-def* **using** *mp all-st-def* **by** *auto*

from *VF* **have** $(t' \mid\!\text{-}\ p,\ s' \mid\!\text{-}\ p) = (\mathit{Fun}\ f\ ss,\ \mathit{Var}\ x)$ *var-ind i x* **by** *auto*
from *main*[*OF this p*(*2*,*1*) *t'*]
obtain *p* **where** $p \in \mathit{poss}\ (t' \cdot \mathit{Var}(z := u))\ p \in \mathit{poss}\ (s' \cdot \mathit{Var}(z := u))\ \mathit{confl}\ (\mathit{Suc}\ i)\ (t' \cdot \mathit{Var}(z := u) \mid\!\text{-}\ p)\ (s' \cdot \mathit{Var}(z := u) \mid\!\text{-}\ p)$
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 $\exists p.\ p \in \mathit{poss}\ (s\ mp \cdot \sigma') \wedge p \in \mathit{poss}\ (t\ mp \cdot \sigma') \wedge \mathit{confl}\ (\mathit{Suc}\ i)\ (s\ mp \cdot \sigma' \mid\!\text{-}\ p)\ (t\ mp \cdot \sigma' \mid\!\text{-}\ p)$
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*: $\bigwedge mp.\ mp \in pp \Longrightarrow \exists p.\ p \in \mathit{poss}\ (s\ mp \cdot \sigma) \wedge p \in \mathit{poss}\ (t\ mp \cdot \sigma) \wedge \mathit{confl}\ n\ (s\ mp \cdot \sigma \mid\!\text{-}\ p)\ (t\ mp \cdot \sigma \mid\!\text{-}\ p)$
by *blast*
define σ' :: (*'f*,*nat* × *'s*,*unit*)*gsubst* **where** $\sigma'\ x = \mathit{conv}'\ (\mathit{Var}\ x)$ **for** x
let $?\sigma = \sigma \circ_s \sigma'$
{
fix x :: *nat* × *'s*
assume ∗: *snd* $x \in S$ ¬ *inf-sort* (*snd x*)
from δ[*THEN sorted-mapD*, *of x*] ∗ **have** $\delta\ x$: *snd x in* $\mathcal{T}(C,\emptyset)$ **by** *auto*
hence *vars*: *vars* $(\delta\ x) = \{\}$ **by** (*simp add*: *hastype-in-Term-empty-imp-vars*)
from ∗ σ[*unfolded cg-subst-ind-def*] **have** $\sigma\ x = \mathit{conv}\ (\delta\ x)$ **by** *blast*
hence $?\sigma\ x = \delta\ x \cdot (\mathit{undefined} \circ_s \sigma')$ **by** (*simp add*: *subst-compose-def conv-def subst-subst*)
also have $\ldots = \delta\ x$ **by** (*rule ground-term-subst*[*OF vars*])
finally have $?\sigma\ x = \delta\ x$ **.**
} **note** $\sigma\delta$ = *this*
have $?\sigma :_s \{x : \iota\ \mathit{in}\ \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C)$
proof (*intro sorted-mapI*, *unfold subst-compose-def hastype-in-restrict-sset conj-imp-eq-imp-imp*)
fix x :: *nat* × *'s* **and** ι
assume $x : \iota\ \mathit{in}\ \mathcal{V}$ **and** $\iota \in S$
then have *snd* $x = \iota$ $\iota \in S$ **by** *auto*
with σ[*unfolded cg-subst-ind-def*, *rule-format*, *of x*]
have $\sigma\ x : \iota\ \mathit{in}\ \mathcal{T}(C,\emptyset)$ **by** *auto*
thus $\sigma\ x \cdot \sigma' : \iota\ \mathit{in}\ \mathcal{T}(C,\emptyset)$ **by** (*rule type-conversion*)
qed
from *pp*[*unfolded wf-pat-complete-iff*[*OF wf*] *match-complete-wrt-def*, *rule-format*, *OF this*]
obtain *mp* μ **where** *mp*: $mp \in pp \cup pp'$ **and** *match*: $\bigwedge ti\ li.\ (ti,\ li) \in mp \Longrightarrow ti \cdot ?\sigma = li \cdot \mu$ **by** *force*
{

```
      assume mp: mp ∈ pp
      from P1[OF this(1)]
      have (s mp, Var (x mp)) ∈ mp (t mp, Var (x mp)) ∈ mp by auto
      from match[OF this(1)] match[OF this(2)] have ident: s mp · ?σ = t mp ·
?σ by auto
      from confl[OF mp] obtain p
        where p: p ∈ poss (s mp · σ) p ∈ poss (t mp · σ) and confl: confl n (s
mp · σ |- p) (t mp · σ |- p)
        by auto
      let ?s = s mp · σ let ?t = t mp · σ
      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) ≠ (g,length
ts) by auto
      define s' where s' = s mp · σ
      define t' where t' = t mp · σ
      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 ∈ 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 ∈ poss
(us2 ! i) and i2: i < length us2 by auto
        from Cons(6)[unfolded s t] i1 i2 have us1 ! i · σ' = us2 ! i · σ' by (auto
simp: list-eq-nth-eq)
        from Cons.IH[OF si(1) ti(1) si(2) ti(2) this]
        show False .
      qed
    }
    with mp have mp: mp ∈ pp' by auto
    show Bex pp' (match-complete-wrt δ)
      unfolding match-complete-wrt-def
    proof (intro bexI[OF - mp] exI[of - μ] ballI, clarify)
      fix ti li
      assume tl: (ti, li) ∈ mp
      have ti · δ = ti · ?σ
      proof (intro term-subst-eq, rule sym, rule σδ)
        fix x
        assume x: x ∈ vars ti
         from *(3) x tl mp show ¬ inf-sort (snd x) by (auto simp: tvars-pat-def
tvars-match-def)
        from *(5) x tl mp show snd x ∈ S
          unfolding wf-pat-def wf-match-def tvars-match-def by auto
```

```
      qed
      also have ... = li · μ using match[OF tl] .
      finally show ti · δ = li · μ .
    qed
  qed
 }
 with easy show ?case by auto
qed

lemma pp-success-pcorrect: pp-success pp ⟹ pat-complete C pp
 by (induct pp rule: pp-success.induct, auto simp: pat-complete-def match-complete-wrt-def)

theorem P-step-set-pcorrect:
  P ⇛s P' ⟹ wf-pats P ⟹ pats-complete C P ⟷ pats-complete C P'
proof (induct P P' rule: P-step-set.induct)
 case (P-fail P)
 with σg show ?case by (auto simp: wf-pats-complete-iff)
next
 case ∗: (P-simp pp pp' P)
 with pp-step-wf have wf-pat pp wf-pats P wf-pats (insert pp P) wf-pats (insert
pp' P)
   by (auto simp: wf-pats-def)
 with pp-step-pcorrect[OF ∗(1)] show ?case
   by (auto simp: wf-pat-complete-iff wf-pats-complete-iff wf-pats-def)
next
 case ∗: (P-remove-pp pp P)
 with pp-step-wf have wf-pat pp wf-pats P wf-pats (insert pp P) by (auto simp:
wf-pats-def)
 then show ?case using pp-success-pcorrect[OF ∗(1)]
   by (auto simp: wf-pats-complete-iff wf-pat-complete-iff)
next
 case ∗: (P-instantiate n pp x P)
 note wfppP = ‹wf-pats (insert pp P)›
 then have wfpp: wf-pat pp and wfP: wf-pats P by (auto simp: wf-pats-def)

 from wfpp ∗(2) have x: snd x ∈ S
   unfolding tvars-pat-def tvars-match-def wf-pat-def wf-match-def by force
 note def = wf-pat-complete-iff[unfolded match-complete-wrt-def]
 define P' where P' = {subst-pat-problem-set τ pp |. τ ∈ τs n x}
 show ?case
   apply (fold P'-def)
 proof (rule ball-insert-un-cong, standard)
   assume complete: Ball P' (pat-complete C)
   show pat-complete C pp unfolding def[OF wfpp]
   proof (intro allI impI)
     fix σ
     assume cg: σ :s {x : ι in V. ι ∈ S} → T(C)
     from sorted-mapD[OF this] x
     have σ x : snd x in T(C) by auto
```

then obtain $f\ ts\ \sigma s$ **where** f: $f : \sigma s \to snd\ x\ in\ C$
and *args*: $ts :_l \sigma s\ in\ \mathcal{T}(C)$
and σx: $\sigma\ x = Fun\ f\ ts$
by (*induct*, *auto*)
from f **have** f: $f : \sigma s \to snd\ x\ in\ C$
by (*meson fun-hastype-def*)
let $?l = length\ ts$
from *args* **have** *len*: $length\ \sigma s = ?l$ **by** (*simp add*: *list-all2-lengthD*)
have l: $?l \leq m$ **using** $m[OF\ f]$ *len* **by** *auto*
have σsS: $\forall \iota \in set\ \sigma s.\ \iota \in S$ **using** *C-sub-S f* **by** *auto*
define σ' **where** $\sigma' = (\lambda\ ys.\ let\ y = fst\ ys\ in\ if\ n \leq y \wedge y < n + ?l \wedge \sigma s\ !\ (y - n) = snd\ ys\ then\ ts\ !\ (y - n)\ else\ \sigma\ ys)$
have *cg*: $\sigma' :_s \{x : \iota\ in\ \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C)$
proof (*intro sorted-mapI*, *unfold hastype-in-restrict-sset conj-imp-eq-imp-imp*)
fix $ys :: nat \times 's$ **and** ι
assume $ys : \iota\ in\ \mathcal{V}$ **and** $\iota \in S$
then have [*simp*]: $\iota = snd\ ys$ **and** *ysS*: $snd\ ys \in S$ **by** *auto*
show $\sigma'\ ys : \iota\ in\ \mathcal{T}(C)$
proof (*cases* $\sigma'\ ys = \sigma\ ys$)
case *True*
thus *?thesis* **using** *cg ysS* **by** (*auto simp*: *sorted-mapD*)
next
case *False*
obtain $y\ s$ **where** *ys*: $ys = (y,s)$ **by** *force*
with *False* **have** y: $y - n < ?l\ n \leq y\ y < n + ?l$ **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*
using σ' *len list-all2-nthD*[*OF args y*(*1*)]
by (*auto simp*: *ys arg*[*symmetric*])
qed
qed
define τ **where** $\tau = subst\ x\ (Fun\ f\ (map\ Var\ (zip\ [n..<n + ?l]\ \sigma s)))$
have $\tau :_s \mathcal{V}\ |`\ tvars\text{-}pat\ pp \to \mathcal{T}(C,\{x : \iota\ in\ \mathcal{V}.\ \iota \in S\})$
using *Fun-hastypeI*[*OF f*, *of* $\{x : \iota\ in\ \mathcal{V}.\ \iota \in S\}$ *map Var* (*zip* $[n..<n + ?l]$ σs)] σsS *wfpp*
by (*auto intro*!: *sorted-mapI*
simp: τ*-def subst-def len*[*symmetric*] *list-all2-conv-all-nth hastype-restrict wf-pat-iff*)
from *wf-pat-subst*[*OF this*]
have *wf2*: *wf-pat* (*subst-pat-problem-set* τ *pp*)**.**
from f **have** $\tau \in \tau s\ n\ x$ **unfolding** τs*-def* τ*-def* τc*-def* **using** *len*[*symmetric*] **by** *auto*
hence *pat-complete C* (*subst-pat-problem-set* τ *pp*) **using** *complete* **by** (*auto simp*: P'*-def*)
from *this*[*unfolded def*[*OF wf2*], *rule-format*, *OF cg*]
obtain *tl* μ **where** *tl*: $tl \in$ *subst-pat-problem-set* τ *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' ∈ pp and tl: tl = {(t' · τ, l) |. (t',l) ∈ tl'} by auto
      show ∃ tl∈ pp. ∃ μ. ∀ (ti, li)∈ tl. ti · σ = li · μ
      proof (intro bexI[OF -  tl'] exI[of - μ], clarify)
        fix ti li
        assume tli: (ti, li) ∈ tl'
        hence tlit: (ti · τ, li) ∈ tl unfolding tl by force
        from match[OF this] have match: ti · τ · σ' = li · μ by auto
        from *(1)[unfolded tvars-disj-pp-def, rule-format, OF tl' tli]
        have vti: fst ‘ vars-term ti ∩ {n..<n + m} = {} by auto
        have ti · σ = ti · (τ ∘s σ')
        proof (rule term-subst-eq, unfold subst-compose-def)
          fix y
          assume y ∈ vars-term ti
          with vti have y: fst y ∉ {n..<n + m} by auto
          show σ y = τ y · σ'
          proof (cases y = x)
            case False
            hence τ y · σ' = σ' y unfolding τ-def subst-def by auto
            also have ... = σ 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 ... = li · μ using match by simp
        finally show ti · σ = li · μ by blast
      qed
    qed
  next
    assume complete: pat-complete C pp
    show ∀ pp ∈ P'. pat-complete C pp
      apply (unfold P'-def)
    proof safe
      fix τ
      assume τ ∈ τs n x
      from this[unfolded τs-def τc-def, simplified]
      obtain f ιs where f: f : ιs → snd x in C and τ: τ = subst x (Fun f (map
Var (zip [n..<n + length ιs] ιs))) by auto
      let ?i = length ιs
      let ?xs = zip [n..<n + length ιs] ιs
      have i: ?i ≤ m by (rule m[OF f])
      have ∀ ι ∈ set ιs. ι ∈ S using C-sub-S f by blast
      with Fun-hastypeI[OF f, of {x : ι in V. ι ∈ S} map Var ?xs] wfpp
      have τ :s V |‘ tvars-pat pp → T(C,{x : ι in V. ι ∈ S})
```

```
    by (auto intro!: sorted-mapI
        simp: τ subst-def hastype-restrict list-all2-conv-all-nth wf-pat-iff)
  note def2 = def[OF wf-pat-subst[OF this]]
  show pat-complete C (subst-pat-problem-set τ pp) unfolding def2
  proof (intro allI impI)
    fix σ assume cg: σ :s {x : ι in V. ι ∈ S} → T(C)
    define σ' where σ' = σ(x := Fun f (map σ ?xs))
    from C-sub-S[OF f] have sortsS: set ιs ⊆ S by auto
    from f have f: f : ιs → snd x in C by (simp add: fun-hastype-def)
    with sorted-mapD[OF cg] set-mp[OF sortsS]
    have Fun f (map σ ?xs) : snd x in T(C)
      by (auto intro!: Fun-hastypeI simp: list-all2-conv-all-nth)
    with sorted-mapD[OF cg]
     have cg: σ' :s {x : ι in V. ι ∈ S} → T(C) by (auto intro!: sorted-mapI
simp: σ'-def)
    from complete[unfolded def[OF wfpp], rule-format, OF this]
    obtain tl μ where tl: tl ∈ pp and tli: ⋀ ti li. (ti, li)∈ tl ⟹ ti · σ' = li ·
μ by force
    from tl have tlm: {(t · τ, l) |. (t,l) ∈ tl} ∈ subst-pat-problem-set τ pp
      unfolding subst-defs-set subst-left-def by auto
    {
      fix ti li
      assume mem: (ti, li) ∈ tl
       from *[unfolded tvars-disj-pp-def] tl mem have vti: fst ‘ vars-term ti ∩
{n..<n + m} = {} by force
      from tli[OF mem] have li · μ = ti · σ' by auto
      also have ... = ti · (τ ∘s σ)
      proof (intro term-subst-eq, unfold subst-compose-def)
        fix y
        assume y ∈ vars-term ti
        with vti have y: fst y ∉ {n..<n + m} by auto
        show σ' y = τ y · σ
        proof (cases y = x)
          case False
          hence τ y · 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 · τ · σ = li · μ by auto
    }
    thus ∃ tl ∈ subst-pat-problem-set τ pp. ∃ μ. ∀ (ti, li)∈tl. ti · σ = li · μ
      by (intro bexI[OF - tlm], auto)
  qed
```

qed
qed
qed
end

Represent a variable-form as a set of maps.

definition *match-of-var-form ff* = {(*Var y*, *Var x*) | *x y*. *y* ∈ *ff x*}

definition *pat-of-var-form ff* = *match-of-var-form* ‘ *ff*

definition *var-form-of-match mp x* = {*y*. (*Var y*, *Var x*) ∈ *mp*}

definition *var-form-of-pat pp* = *var-form-of-match* ‘ *pp*

definition *tvars-var-form-pat ff* = (⋃*f* ∈ *ff*. ⋃(*range f*))

definition *var-form-match* **where**
var-form-match mp ⟷ *mp* ⊆ *range* (*map-prod Var Var*)

definition *var-form-pat pp* ≡ ∀ *mp* ∈ *pp*. *var-form-match mp*

lemma *match-of-var-form-of-match*:
assumes *var-form-match mp*
shows *match-of-var-form* (*var-form-of-match mp*) = *mp*
using *assms*
by (*auto simp*: *var-form-match-def match-of-var-form-def var-form-of-match-def*)

lemma *tvars-match-var-form*:
assumes *var-form-match mp*
shows *tvars-match mp* = {*v*. ∃ *x*. (*Var v*, *Var x*) ∈ *mp*}
using *assms* **by** (*force simp*: *var-form-match-def tvars-match-def*)

lemma *pat-of-var-form-pat*:
assumes *var-form-pat pp*
shows *pat-of-var-form* (*var-form-of-pat pp*) = *pp*
using *assms match-of-var-form-of-match*
by (*auto simp*: *var-form-pat-def var-form-of-pat-def pat-of-var-form-def*)

lemma *tvars-pat-var-form*: *tvars-pat* (*pat-of-var-form ff*) = *tvars-var-form-pat ff*
by (*fastforce simp*: *tvars-var-form-pat-def tvars-pat-def tvars-match-def pat-of-var-form-def match-of-var-form-def*
split: *prod.splits*)

lemma *tvars-var-form-pat*:
assumes *var-form-pat pp*
shows *tvars-var-form-pat* (*var-form-of-pat pp*) = *tvars-pat pp*
apply (*subst*(*2*) *pat-of-var-form-pat*[*OF assms*,*symmetric*])
by (*simp add*: *tvars-pat-var-form*)

lemma *pat-complete-var-form*:
pat-complete C (*pat-of-var-form ff*) ⟷
(∀ σ :$_s$ $\mathcal{V}$ |‘ *tvars-var-form-pat ff* → $\mathcal{T}$(*C*). ∃ *f* ∈ *ff*. ∃ μ. ∀ *x*. ∀ *y* ∈ *f x*. σ *y* = μ *x*)
proof−
define *V* **where** *V* = $\mathcal{V}$ |‘ *tvars-var-form-pat ff*
have *boo*: $\mathcal{V}$ |‘ *tvars-pat* {{(*Var* (*a*, *b*), *Var xa*) | *xa a b*. (*a*, *b*) ∈ *x xa*} |. *x* ∈ *ff*} = *V*
apply (*unfold V-def*)
apply (*subst tvars-pat-var-form*[*of ff*, *symmetric*])
by (*auto simp*: *V-def pat-of-var-form-def match-of-var-form-def*)
show *?thesis*
apply (*fold V-def*)
apply (*auto simp*: *pat-complete-def match-complete-wrt-def pat-of-var-form-def match-of-var-form-def imp-conjL imp-ex boo*)
apply (*metis old.prod.exhaust*)
by *metis*
qed

lemma *pat-complete-var-form-set*:
pat-complete C (*pat-of-var-form ff*) ⟷
(∀ σ :$_s$ $\mathcal{V}$ |‘ *tvars-var-form-pat ff* → $\mathcal{T}$(*C*). ∃ *f* ∈ *ff*. ∃ μ. ∀ *x*. σ ‘ *f x* ⊆ {μ *x*})
by (*auto simp*: *pat-complete-var-form image-subset-iff*)

lemma *pat-complete-var-form-Uniq*:
pat-complete C (*pat-of-var-form ff*) ⟷
(∀ σ :$_s$ $\mathcal{V}$ |‘ *tvars-var-form-pat ff* → $\mathcal{T}$(*C*). ∃ *f* ∈ *ff*. ∀ *x*. *UNIQ* (σ ‘ *f x*))
proof−
{ fix σ *f* **assume** σ: σ :$_s$ $\mathcal{V}$ |‘ *tvars-var-form-pat ff* → $\mathcal{T}$(*C*) **and** *f*: *f* ∈ *ff*
have (∃ μ. ∀ *x*. σ ‘ *f x* ⊆ {μ *x*}) ⟷ (∀ *x*. ∃$_{\leq 1}$ *y*. *y* ∈ σ ‘ *f x*)
proof (*safe*)
fix μ *x*
assume ∀ *x*. σ ‘ *f x* ⊆ {μ *x*}
from *this*[*rule-format*, *of x*]
have *y* ∈ *f x* ⟹ σ *y* = μ *x* **for** *y* **by** *auto*
then show ∃$_{\leq 1}$ *y*. *y* ∈ σ ‘ *f x* **by** (*auto intro*!: *Uniq-I*)
next
define μ **where** μ *x* = *the-elem* (σ ‘ *f x*) **for** *x*
fix *x* **assume** ∀ *x*. ∃$_{\leq 1}$ *y*. *y* ∈ σ ‘ *f x*
from *Uniq-eq-the-elem*[*OF this*[*rule-format*], *folded* μ-*def*]
show ∃ μ. ∀ *x*. σ ‘ *f x* ⊆ {μ *x*} **by** *auto*
qed
}
then show *?thesis* **by** (*simp add*: *pat-complete-var-form-set*)
qed

lemma *ex-var-form-pat*: (∃ *f*∈*var-form-of-pat pp*. *P f*) ⟷ (∃ *mp* ∈ *pp*. *P* (*var-form-of-match mp*))
by (*auto simp*: *var-form-of-pat-def*)

```
lemma pat-complete-var-form-nat:
  assumes fin: ∀(x,ι) ∈ tvars-var-form-pat ff. finite-sort C ι
    and uniq: ∀f ∈ ff. ∀x::'v. UNIQ (snd ‘ f x)
  shows pat-complete C (pat-of-var-form ff) ⟷
  (∀α. (∀v ∈ tvars-var-form-pat ff. α v < card-of-sort C (snd v)) ⟶
  (∃f ∈ ff. ∀x. UNIQ (α ‘ f x)))
  (is ?l ⟷ (∀α. ?s α ⟶ ?r α))
proof safe
  note fin = fin[unfolded Ball-Pair-conv, rule-format]
  { fix α
    assume l: ?l and a: ?s α
    define σ :: - ⇒ (-,unit) term where
      σ ≡ λ(x,ι). term-of-index C ι (α (x,ι))
    have σ (x,ι) : ι in 𝒯(C) if x: (x,ι) ∈ tvars-var-form-pat ff for x ι
      using term-of-index-bij[OF fin, OF x]
        a[unfolded Ball-Pair-conv, rule-format, OF x]
      by (auto simp: bij-betw-def σ-def)
    then have σ :s 𝒱 |‘ tvars-var-form-pat ff → 𝒯(C)
      by (auto intro!: sorted-mapI simp: hastype-restrict)
    from l[unfolded pat-complete-var-form-Uniq, rule-format, OF this]
    obtain f where f: f ∈ ff and u: ⋀x. UNIQ (σ ‘ f x) by auto
    have id: y ∈ f x ⟹ index-of-term C (σ y) = α y for y x
      using assms a f
      by (force simp: σ-def index-of-term-of-index tvars-var-form-pat-def Ball-def
split: prod.splits)
    then have α ‘ f x = index-of-term C ‘ σ ‘ f x for x
      by (auto simp: image-def)
    then have UNIQ (α ‘ f x) for x by (simp add: image-Uniq[OF u])
    with f show ?r α by auto
  next
    assume r: ∀α. ?s α ⟶ ?r α
    show ?l
      unfolding pat-complete-var-form-Uniq
    proof safe
      fix σ
      assume σ: σ :s 𝒱 |‘ tvars-var-form-pat ff → 𝒯(C)
      from sorted-mapD[OF this]
      have ty: (x,ι) ∈ tvars-var-form-pat ff ⟹ σ (x,ι) : ι in 𝒯(C)
        for x ι by (auto simp: hastype-restrict)
      define α where α ≡ index-of-term C ∘ σ
      have α (x,ι) < card-of-sort C ι if x: (x,ι) ∈ tvars-var-form-pat ff
        for x ι using index-of-term-bij[OF fin[OF x]] ty[OF x]
        by (auto simp: α-def bij-betw-def)
      then have ∃f∈ff. ∀x. UNIQ (α ‘ f x) by (auto intro!: r[rule-format])
      then obtain f where f: f ∈ ff and u: ⋀x. UNIQ (α ‘ f x) by auto
      have UNIQ (σ ‘ f x) for x
      proof−
        from uniq[rule-format, OF f]
```

```
      have ex: ∃ ι. snd ‘ f x ⊆ {ι}
        by (auto simp: subset-singleton-iff-Uniq)
      then obtain ι where sub: snd ‘ f x ⊆ {ι} by auto
      { fix y κ assume yk: (y,κ) ∈ f x
        with sub have [simp]: κ = ι by auto
        from yk f have y: (y,ι) ∈ tvars-var-form-pat ff
          by (auto simp: tvars-var-form-pat-def)
        from y fin[OF y]
        have term-of-index C ι (α (y,κ)) = σ (y,κ)
          by (auto simp: α-def hastype-restrict
              intro!: term-of-index-of-term sorted-mapD[OF σ])
      }
      then have y ∈ f x ⟹ term-of-index C ι (α y) = σ y for y
        by (cases y, auto)
      then have σ ‘ f x = term-of-index C ι ‘ α ‘ f x
        by (auto simp: image-def)
      then show UNIQ (σ ‘ f x) by (simp add: image-Uniq[OF u])
    qed
    with f show ∃ f ∈ ff. ∀ x. UNIQ (σ ‘ f x) by auto
  qed
 }
qed
```

A problem is in finite variable form, if only variables occur in the problem and these variable all have a finite sort. Moreover, comparison of variables is only done if they have the same sort.

```
definition finite-var-form-match :: ('f,'s) ssig ⇒ ('f,'v,'s)match-problem-set ⇒
bool where
 finite-var-form-match C mp ⟷ var-form-match mp ∧
 (∀ l x y. (Var x, l) ∈ mp ⟶ (Var y, l) ∈ mp ⟶ snd x = snd y) ∧
 (∀ l x. (Var x, l) ∈ mp ⟶ finite-sort C (snd x))

lemma finite-var-form-matchD:
 assumes finite-var-form-match C mp and (t,l) ∈ mp
 shows ∃ x ι y. t = Var (x,ι) ∧ l = Var y ∧ finite-sort C ι ∧
   (∀ z. (Var z, Var y) ∈ mp ⟶ snd z = ι)
 using assms by (auto simp: finite-var-form-match-def var-form-match-def)

definition finite-var-form-pat :: ('f,'s) ssig ⇒ ('f,'v,'s)pat-problem-set ⇒ bool where
 finite-var-form-pat C p = (∀ mp ∈ p. finite-var-form-match C mp)

lemma finite-var-form-patD:
 assumes finite-var-form-pat C pp mp ∈ pp (t,l) ∈ mp
 shows ∃ x ι y. t = Var (x,ι) ∧ l = Var y ∧ finite-sort C ι ∧
   (∀ z. (Var z, Var y) ∈ mp ⟶ snd z = ι)
 using assms[unfolded finite-var-form-pat-def] finite-var-form-matchD by metis

lemma finite-var-form-imp-of-var-form-pat:
 finite-var-form-pat C pp ⟹ var-form-pat pp
```

by (*auto simp*: *finite-var-form-pat-def var-form-pat-def finite-var-form-match-def*)

context *pattern-completeness-context* **begin**

definition *weak-finite-var-form-match* :: ('*f*,'*v*,'*s*)*match-problem-set* ⇒ *bool* **where**
weak-finite-var-form-match mp = ((∀ (*t*,*l*) ∈ *mp*. ∃ *y*. *l* = *Var y*)
∧ (∀ *f ts y*. (*Fun f ts*, *Var y*) ∈ *mp* ⟶
(∃ *x*. (*Var x*, *Var y*) ∈ *mp* ∧ *inf-sort* (*snd x*))
∧ (∀ *t*. (*t*, *Var y*) ∈ *mp* ⟶ *root t* ∈ {*None*, *Some* (*f*,*length ts*)})))

definition *weak-finite-var-form-pat* :: ('*f*,'*v*,'*s*)*pat-problem-set* ⇒ *bool* **where**
weak-finite-var-form-pat p = (∀ *mp* ∈ *p*. *weak-finite-var-form-match mp*)

end

lemma *finite-var-form-pat-UNIQ-sort*:
assumes *fvf*: *finite-var-form-pat C pp*
and *f*: *f* ∈ *var-form-of-pat pp*
shows *UNIQ* (*snd* ‘ *f x*)
proof (*intro Uniq-I*, *clarsimp*)
from *f* **obtain** *mp* **where** *mp*: *mp* ∈ *pp* **and** *f*: *f* = *var-form-of-match mp*
by (*auto simp*: *var-form-of-pat-def*)
fix *y ι z κ* **assume** (*y*,*ι*) ∈ *f x* (*z*,*κ*) ∈ *f x*
with *f* **have** *y*: (*Var* (*y*,*ι*), *Var x*) ∈ *mp* **and** *z*: (*Var* (*z*,*κ*), *Var x*) ∈ *mp*
by (*auto simp*: *var-form-of-match-def*)
from *finite-var-form-patD*[*OF fvf mp y*] *z*
show *ι* = *κ* **by** *auto*
qed

lemma *finite-var-form-pat-pat-complete*:
assumes *fvf*: *finite-var-form-pat C pp*
shows *pat-complete C pp* ⟷
(∀ *α*. (∀ *v* ∈ *tvars-pat pp*. *α v* < *card-of-sort C* (*snd v*)) ⟶
(∃ *mp* ∈ *pp*. ∀ *x*. *UNIQ* {*α y* |*y*. (*Var y*, *Var x*) ∈ *mp*}))
proof−
note *vf* = *finite-var-form-imp-of-var-form-pat*[*OF fvf*]
note *pat-complete-var-form-nat*[*of var-form-of-pat pp C*]
note *this*[*unfolded tvars-var-form-pat*[*OF vf*]]
note ∗ = *this*[*unfolded pat-of-var-form-pat*[*OF vf*]]
show *?thesis*
apply (*subst* ∗)
subgoal
proof
fix *y ι*
assume *y*: (*y*,*ι*) ∈ *tvars-pat pp*
from *y* **obtain** *mp t l* **where** *mp*: *mp* ∈ *pp* **and** *tl*:(*t*,*l*) ∈ *mp* **and** *yt*: (*y*, *ι*)
∈ *vars t*
by (*auto simp*: *tvars-pat-def tvars-match-def*)
from *finite-var-form-patD*[*OF fvf mp tl*] *yt*

show *finite-sort C ι* **by** *auto*
qed
subgoal using *finite-var-form-pat-UNIQ-sort*[*OF fvf*] **by** *force*
subgoal
apply (*rule all-cong*)
apply (*unfold ex-var-form-pat*)
apply (*rule bex-cong*[*OF refl*])
apply (*rule all-cong1*)
apply (*rule arg-cong*[*of - - UNIQ*])
by (*auto simp*: *var-form-of-match-def*)
done
qed

end

7 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

7.1 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* ⇒ $('f,'v,'s)$*match-problem-set*

where *mp-mset* ≡ *set-mset*

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

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

where *pats-mset* ≡ *image pat-mset o set-mset*

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

context *pattern-completeness-context*
begin

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

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

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

definition τ*s-list* :: *nat* ⇒ *nat* × *'s* ⇒ (*'f*,*nat* × *'s*)*subst list* **where**
τ*s-list n x* = *map* (τ*c n x*) (*Cl* (*snd x*))

inductive *mp-step-mset* :: (*'f*,*'v*,*'s*)*match-problem-mset* ⇒ (*'f*,*'v*,*'s*)*match-problem-mset* ⇒ *bool* (**infix** ‹→$_m$› *50*)**where**
match-decompose: (*f*,*length ts*) = (*g*,*length ls*)
⟹ *add-mset* (*Fun f ts*, *Fun g ls*) *mp* →$_m$ *mp* + *mset* (*zip ts ls*)
| *match-match*: *x* ∉ ⋃ (*vars* ‘ *snd* ‘ *set-mset mp*)
⟹ *add-mset* (*t*, *Var x*) *mp* →$_m$ *mp*
| *match-duplicate*: *add-mset pair* (*add-mset pair mp*) →$_m$ *add-mset pair mp*
| *match-decompose'*: *mp* + *mp'* →$_m$ (∑ (*t*, *l*) ∈# *mp*. *mset* (*zip* (*args t*) (*map Var ys*))) + *mp'*
if ⋀ *t l*. (*t*,*l*) ∈# *mp* ⟹ *l* = *Var y* ∧ *root t* = *Some* (*f*,*n*)
⋀ *t l*. (*t*,*l*) ∈# *mp'* ⟹ *y* ∉ *vars l*
lvars-disj-mp ys (*mp-mset* (*mp* + *mp'*)) *length ys* = *n*
size mp ≥ *2*
improved

inductive *match-fail* :: (*'f*,*'v*,*'s*)*match-problem-mset* ⇒ *bool* **where**
match-clash: (*f*,*length ts*) ≠ (*g*,*length ls*)
⟹ *match-fail* (*add-mset* (*Fun f ts*, *Fun g ls*) *mp*)
| *match-clash'*: *Conflict-Clash s t* ⟹ *match-fail* (*add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp*))
| *match-clash-sort*: $\mathcal{T}$(*C*,$\mathcal{V}$) *s* ≠ $\mathcal{T}$(*C*,$\mathcal{V}$) *t* ⟹ *match-fail* (*add-mset* (*s*, *Var x*) (*add-mset* (*t*, *Var x*) *mp*))

inductive *pp-step-mset* :: (*'f*,*'v*,*'s*)*pat-problem-mset* ⇒ (*'f*,*'v*,*'s*)*pats-problem-mset* ⇒ *bool*
(**infix** ‹⇒$_m$› *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*)
$\wedge$ (*improved* $\longrightarrow$ *s* = *Var x* $\wedge$ *is-Fun t*) $\Longrightarrow$
add-mset mp pp $\Rightarrow_m$ *mset* (*map* (λ τ. *subst-pat-problem-mset* τ (*add-mset mp pp*)) (τ*s-list n x*))
| *pat-inf-var-conflict*: *Ball* (*pat-mset pp*) *inf-var-conflict* $\Longrightarrow$ *pp* $\neq$ {#}
$\Longrightarrow$ *Ball* (*tvars-pat* (*pat-mset pp'*)) (λ *x*. $\neg$ *inf-sort* (*snd x*)) $\Longrightarrow$
($\neg$ *improved* $\Longrightarrow$ *pp'* = {#})
$\Longrightarrow$ *pp* + *pp'* $\Rightarrow_m$ {# *pp'* #}

inductive *pat-fail* :: ('*f*,'*v*,'*s*)*pat-problem-mset* $\Rightarrow$ *bool* **where**
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'*}

7.2 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 ∈ p' (t,l) ∈ mp
    hence sub: fst ‘ vars t ⊆ V unfolding V-def by force
    {
      fix x
      assume x ∈ fst ‘ vars t
      with sub have x ∈ V by auto
      with ‹finite V› have x ≤ Max V by simp
      also have ... < n unfolding n-def by simp
      finally have x < n .
    }
    hence fst ‘ vars t ∩ {n..<n + m} = {} by force
  }
  thus ?thesis unfolding tvars-disj-pp-def p'-def[symmetric]
    by (intro exI[of - n] ballI, force)
qed

lemma mp-mset-in-pat-mset: mp ∈# pp ⟹ mp-mset mp ∈ pat-mset pp
  by auto
lemma mp-step-mset-cong:
  assumes (→m)** mp mp'
  shows (add-mset (add-mset mp p) P, add-mset (add-mset mp' p) P) ∈ ⇛*
  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) ∈ P-step
    unfolding P-step-def by auto
  with step(3)
  show ?case by simp
qed auto

lemma mp-step-mset-vars: assumes mp →m mp'
  shows tvars-match (mp-mset mp) ⊇ tvars-match (mp-mset mp')
  using assms
proof induct
  case ∗: (match-decompose' mp y f n mp' ys)
  {
    let ?mset = mset :: - ⇒ ('f,'v,'s)match-problem-mset
    fix x
    assume x ∈ tvars-match (mp-mset ((∑(t, l)∈#mp. ?mset (zip (args t) (map
Var ys)))))
    from this[unfolded tvars-match-def, simplified]
    obtain t l ti yi where tl: (t,l) ∈# mp and tiyi: (ti,yi) ∈# ?mset (zip (args t)
(map Var ys))
      and x: x ∈ vars ti
      by auto
    from ∗(1)[OF tl] obtain ts where l: l = Var y and t: t = Fun f ts and lts:
length ts = n
      by (cases t, auto)
```

```
    from tiyi[unfolded t] have ti ∈ set ts
      using set-zip-leftD by fastforce
    with x t have x ∈ vars t by auto
    hence x ∈ tvars-match (mp-mset mp) using tl unfolding tvars-match-def by
auto
  }
  thus ?case unfolding tvars-match-def by force
qed (auto simp: tvars-match-def set-zip)

lemma mp-step-mset-steps-vars: assumes (→m)** mp mp'
  shows tvars-match (mp-mset mp) ⊇ tvars-match (mp-mset mp')
  using assms by (induct, insert mp-step-mset-vars, auto)

end

context pattern-completeness-context-with-assms begin

lemma pat-fail-or-trans-or-finite-var-form:
  fixes p :: ('f,'v,'s) pat-problem-mset
  assumes improved ⟹ infinite (UNIV :: 'v set) and wf: wf-pat (pat-mset p)
 shows pat-fail p ∨ (∃ ps. p ⇒m ps) ∨ (improved ∧ finite-var-form-pat C (pat-mset
p))
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 {#} ∈# 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 ∃ mp s t. mp ∈# p ∧ (s,t) ∈# mp ∧ is-Fun t)
      case True
     then obtain mp s t where mp: mp ∈# p and (s,t) ∈# mp and is-Fun t by
auto
      then obtain g ts where mem: (s,Fun g ts) ∈# 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*) ∈# *mp* ∧ *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*: ⋀ *mp s l*. *mp* ∈# *p* ⟹ (*s*,*l*) ∈# *mp* ⟹ *is-Var l* **by** *auto*
let *?single-var* = (∃ *mp t x*. *add-mset* (*t*,*Var x*) *mp* ∈# *p* ∧ *x* ∉ ⋃ (*vars* ‘ *snd* ‘ *set-mset mp*))
let *?duplicate* = (∃ *mp pair*. *add-mset pair* (*add-mset pair mp*) ∈# *p*)
show *?thesis*
proof (*cases ?single-var* ∨ *?duplicate*)
case *True*
thus *?thesis*
proof
assume *?single-var*
then obtain *mp t x* **where** *mp*: *add-mset* (*t*,*Var x*) *mp* ∈# *p* **and** *x*: *x* ∉ ⋃ (*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*) ∈# *p* (**is** *?dup* ∈# *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* ∈# *p*
with *empty-p* **have** *mp-e*: *mp* ≠ {#} **by** *auto*
obtain *s l* **where** *sl*: (*s*,*l*) ∈# *mp* **using** *mp-e* **by** *auto*
from *rhs-vars*[*OF mpp sl*] *sl* **obtain** *x* **where** *sx*: (*s*, *Var x*) ∈# *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'*: $mp' = add\text{-}mset\ (t,\ Var\ x)\ mp''$ **by** (*rule mset-add*)
from *ndup*[*simplified*, *rule-format*] *mpp* **have** $s \neq t$ **unfolding** *mp mp'* **by** *auto*
hence $\exists\ s\ t\ x\ mp'.\ mp = add\text{-}mset\ (s,\ Var\ x)\ (add\text{-}mset\ (t,\ Var\ x)\ mp') \wedge s \neq t$ **unfolding** *mp mp'* **by** *auto*
} **note** *two* = *this*
show *?thesis*
proof (*cases* $\exists\ mp\ s\ t\ x.\ add\text{-}mset\ (s,\ Var\ x)\ (add\text{-}mset\ (t,\ Var\ x)\ mp) \in\# p \wedge Conflict\text{-}Clash\ s\ t$)
case *True*
then obtain *mp s t x* **where**
mp: $add\text{-}mset\ (s,\ Var\ x)\ (add\text{-}mset\ (t,\ Var\ x)\ mp) \in\# p$ (**is** *?mp* $\in\#$ -) **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 (*cases* $\exists\ mp\ s\ t\ x\ y.\ add\text{-}mset\ (s,\ Var\ x)\ (add\text{-}mset\ (t,\ Var\ x)\ mp) \in\# p \wedge Conflict\text{-}Var\ s\ t\ y \wedge \neg\ inf\text{-}sort\ (snd\ y)$)
case *True*
show *?thesis*
proof (*cases improved*)
case *not-impr*: *False*
from *True* **obtain** *mp s t x y* **where**
mp: $add\text{-}mset\ (s,\ Var\ x)\ (add\text{-}mset\ (t,\ Var\ x)\ mp) \in\# p$ (**is** *?mp* $\in\#$ -) **and** *conf*: *Conflict-Var s t y* **and** *y*: $\neg\ inf\text{-}sort\ (snd\ y)$
by *blast*
from *mp* **obtain** p' **where** *p*: $p = add\text{-}mset\ ?mp\ p'$ **by** (*rule mset-add*)
let *?mp* = $add\text{-}mset\ (s,\ Var\ x)\ (add\text{-}mset\ (t,\ Var\ x)\ mp)$
from *pat-instantiate*[*OF - disjI2*, *of n ?mp p' s x t y*, *folded p*, *OF fresh*]
show *?thesis* **using** *y conf not-impr* **by** *auto*
next
case *impr*: *True*

have $(pat\text{-}fail\ p \vee (\exists\ ps.\ p \Rightarrow_m ps)) \vee weak\text{-}finite\text{-}var\text{-}form\text{-}pat\ (pat\text{-}mset\ p)$
proof (*cases weak-finite-var-form-pat* (*pat-mset p*))
case *False*
from *this*[*unfolded weak-finite-var-form-pat-def*] **obtain** *mp*
where *mp*: *mp* $\in\#$ *p* **and** *nmp*: $\neg\ weak\text{-}finite\text{-}var\text{-}form\text{-}match\ (mp\text{-}mset\ mp)$ **by** *auto*
from *mset-add*[*OF mp*] **obtain** p' **where** *p'*: $p = add\text{-}mset\ mp\ p'$ **by**

auto
from *rhs-vars*[*OF mp*] **have** ((∀ (*t*, *l*)∈#*mp*. ∃ *y*. *l* = *Var y*) ∧ *b*) = *b* **for** *b*
by *force*
note *nmp* = *nmp*[*unfolded weak-finite-var-form-match-def this*]
from *this*[*simplified*] **obtain** *f ss y* **where**
s: (*Fun f ss*, *Var y*) ∈# *mp* **and**
violation: ((∀ *x*. (*Var x*, *Var y*) ∈# *mp* ⟶ ¬ *inf-sort* (*snd x*)) ∨
(∃ *t g n*. (*t*, *Var y*) ∈# *mp* ∧ *root t* = *Some* (*g*, *n*) ∧ *root t* ≠ *Some* (*f*, *length ss*)))
(**is** *?A* ∨ *?B*)
by *force*
let *?s* = *Fun f ss*
let *?n* = *length ss*
show *?thesis*
proof (*cases ?B*)
case *True*
then obtain *t g n* **where** *t*: (*t*, *Var y*) ∈# *mp root t* = *Some* (*g*, *n*) *root t* ≠ *Some* (*f*, *?n*)
by *auto*
from *t* **have** *st*: (*?s*, *Var y*) ≠ (*t*, *Var y*) **by** (*cases t*, *auto*)
define *mp′* **where** *mp′* = *mp* − {#(*?s*, *Var y*),(*t*, *Var y*)#}
from *s t*(*1*) *st* **have** *mp* = *add-mset* (*?s*, *Var y*) (*add-mset* (*t*, *Var y*) *mp′*)
unfolding *mp′-def*
by (*metis Multiset.diff-add add-mset-add-single diff-union-swap insert-DiffM*)
with *no-clash mp* **have** ¬ *Conflict-Clash ?s t* **by** *metis*
moreover have *Conflict-Clash ?s t*
using *t* **by** (*cases t*, *auto simp*: *conflicts.simps*)
ultimately show *?thesis* **..**
next
case *no-clash′*: *False*
with *violation* **have** *finsort*: ⋀ *x*. (*Var x*, *Var y*) ∈# *mp* ⟹ ¬ *inf-sort* (*snd x*) **by** *blast*
show *?thesis*
proof (*cases* ∃ *x*. (*Var x*, *Var y*) ∈# *mp*)
case *True*
then obtain *x* **where** *t*: (*Var x*, *Var y*) ∈# *mp* (**is** (*?t*,-) ∈# -) **by** *auto*
from *finsort*[*OF t*] **have** *fin*: ¬ *inf-sort* (*snd x*) **.**
from *s t fin pat-instantiate*[*OF - disjI2*, *of - mp p′ ?t y ?s x*, *folded p′*, *OF fresh*]
show *?thesis* **by** (*auto simp*: *conflicts.simps*)
next
case *False*
define *test-y* **where** *test-y tl* = (*snd tl* = *Var y*) **for** *tl* :: (*′f*, *nat* × *′s*) *term* × (*′f*,*′v*)*term*
define *mpy* **where** *mpy* = *filter-mset test-y mp*

```
                have size: size mpy ≥ 2
                proof −
                  from mset-add[OF s] obtain mp′ where mp′: mp = add-mset
(?s, Var y) mp′ by blast
                  have y ∈ ⋃ (vars ‘ snd ‘ mp-mset mp′)
                    using nsvar[rule-format] mp′ mp by blast
                  then obtain t′ l′ where tl′: (t′,l′) ∈# mp′ and y ∈ vars l′ by
auto
                  with rhs-vars[OF mp, of t′ l′] mp′ have is-Var l′ by auto
                  with ‹y ∈ vars l′› have l′: l′ = Var y by auto
                  hence (t′,Var y) ∈# mp′ using tl′ by auto
                from mset-add[OF this] obtain mp′′ where mp′′: mp′ = add-mset
(t′, Var y) mp′′
                  by auto
                  have mpy: mpy = add-mset (?s, Var y) (add-mset (t′, Var y)
(filter-mset test-y mp′′))
                    unfolding mpy-def mp′ mp′′ by (simp add: test-y-def)
                  thus ?thesis by simp
                qed
                define mpny where mpny = filter-mset (Not o test-y) mp
                have id: mp = mpy + mpny by (simp add: mpy-def mpny-def)
                {
                  fix t l
                  assume (t, l) ∈# mpny
                    hence l ≠ Var y (t,l) ∈# mp unfolding mpny-def test-y-def
o-def by auto
                  with rhs-vars[OF mp, of t l] have y ∉ vars l by (cases l, auto)
                } note mpny = this
                {
                  fix t l
                  assume (t, l) ∈# mpy
                    hence l: l = Var y and pair: (t,Var y) ∈# mp unfolding
mpy-def test-y-def o-def by auto
                  with False obtain g ts where t: t = Fun g ts by (cases t, auto)
                  from no-clash′ pair t have root t = Some (f,?n) by auto
                  with l have l = Var y ∧ root t = Some (f,?n) by auto
                } note mpy = this
                define VV where VV = ⋃ (vars ‘ snd ‘ mp-mset mp)
                have finite VV by (auto simp: VV-def)
                with assms(1)[OF impr] have infinite (UNIV − VV) by auto
                  then obtain Ys where Ys: Ys ⊆ UNIV − VV card Ys = ?n
finite Ys
                  by (meson infinite-arbitrarily-large)
                from Ys(2−3) obtain ys where ys: distinct ys length ys = ?n
set ys = Ys
                  by (metis distinct-card finite-distinct-list)
                with Ys have dist: VV ∩ set ys = {} by auto
                have lvars-disj-mp ys (mp-mset mp) length ys = ?n
                  unfolding lvars-disj-mp-def using ys dist unfolding VV-def
```

by *auto*
from *match-decompose'*[*of mpy y f ?n mpny*, *folded id*, *OF mpy mpny this size impr*]
obtain *mp'* **where** *mp* $\rightarrow_m$ *mp'* **by** *force*
from *pat-simp-mp*[*OF this*, *of p'*] *p'* **show** *?thesis* **by** *auto*
qed
qed
qed *auto*

thus *?thesis*
proof (*elim context-disjE*)
assume *no-step*: ¬ (*pat-fail p* ∨ (∃ *ps*. *p* $\Rightarrow_m$ *ps*))
assume *weak-finite-var-form-pat* (*pat-mset p*)
note *wfvf* = *this*[*unfolded weak-finite-var-form-pat-def weak-finite-var-form-match-def*, *rule-format*]
note *get-var* = *wfvf*[*THEN conjunct1*, *rule-format*]
note *fun-case* = *wfvf*[*THEN conjunct2*, *rule-format*]

define *fin* **where** *fin mp* = *Ball* (*tvars-match* (*mp-mset mp*)) (λ *x*. ¬ *inf-sort* (*snd x*)) **for** *mp* :: ('*f*,'*v*,'*s*) *match-problem-mset*
define *p-fin* **where** *p-fin* = *filter-mset fin p*
define *p-inf* **where** *p-inf* = *filter-mset* (*Not o fin*) *p*
have *p-split*: *p* = *p-inf* + *p-fin* **unfolding** *p-fin-def p-inf-def* **by** *auto*
show *?thesis*
proof (*cases p-inf* = {#})
case *True*
have *fin*: ⋀ *mp*. *mp* ∈# *p* ⟹ *fin mp* **unfolding** *p-split True* **unfolding** *p-fin-def* **by** *auto*
have *finite-var-form-pat C* (*pat-mset p*)
unfolding *finite-var-form-pat-def finite-var-form-match-def var-form-match-def*
proof (*intro ballI conjI subsetI allI impI*, *clarify*)
fix *mp l*
assume *mp*: *mp* ∈ *pat-mset p*
{ fix *t* **assume** *tl*: (*t*,*l*) ∈ *mp*
from *get-var*[*OF mp tl*] *tl* **obtain** *y* **where**
ty: (*t*, *Var y*) ∈ *mp* **and** *ly*: *l* = *Var y* **by** (*cases l*, *auto*)
have *is-Var t*
proof (*cases t*)
case (*Fun f ts*)
with *ty* **have** (*Fun f ts*, *Var y*) ∈ *mp* **by** *auto*
from *fun-case*[*OF - this*] *mp* **obtain** *x* **where** (*Var x*, *Var y*) ∈ *mp inf-sort* (*snd x*) **by** *auto*
with *fin*[*unfolded fin-def tvars-match-def*] *mp tl* **have** *False* **by** *auto*
thus *?thesis* **by** *auto*
qed *auto*
with *ly* **show** (*t*,*l*) ∈ *range* (*map-prod Var Var*) **by** *auto*
} note *var-var* = *this*

fix x **assume** xl: $(Var\ x,\ l) \in mp$
then have xmp: $x \in tvars\text{-}match\ mp$ **by** (*force simp*: *tvars-match-def*)
with wf[*unfolded wf-pat-def wf-match-def*, *rule-format*, *OF mp*]
have sxS: $snd\ x \in S$ **by** *auto*
from *mp xmp fin fin-def* **have** $\neg$*inf-sort* (*snd x*) **by** *auto*
with *inf-sort*[*OF sxS*]
show *fint*: *finite-sort C* (*snd x*) **by** *auto*
fix y **assume** yl: $(Var\ y,\ l) \in mp$
from *yl var-var* **obtain** z **where** l: $l = Var\ z$ **by** *force*
show $snd\ x = snd\ y$
proof (*cases* $x = y$)
case *False*
from *mp* **obtain** mp' **where** mp': $mp' \in\# p$ **and** mp: $mp = mp\text{-}mset\ mp'$ **by** *auto*
from *False xl yl* **obtain** mp''
where $mp' = add\text{-}mset\ (Var\ x,\ Var\ z)\ (add\text{-}mset\ (Var\ y,\ Var\ z)\ mp'')$
unfolding *l mp* **by** (*metis insert-DiffM insert-noteq-member prod.inject term.inject*(*1*))
with *no-clash mp'* **have** $\neg$ *Conflict-Clash* (*Var x*) (*Var y*)
by (*metis conflicts.simps*(*1*))
thus $snd\ x = snd\ y$ **by** (*simp add*: *conflicts.simps split*: *if-splits*)
qed *auto*
qed
with *impr* **show** *?thesis* **by** *auto*
next
case *False*
have $\forall\, x{\in}tvars\text{-}pat\ (pat\text{-}mset\ p\text{-}fin).\ \neg\ inf\text{-}sort\ (snd\ x)$ **unfolding** *p-fin-def fin-def*
by (*auto simp*: *tvars-pat-def*)
from *pat-inf-var-conflict*[*OF - False this*, *folded p-split*] *no-step*
obtain *mp* **where** *mp*: $mp \in\# p$ **and** *inf*: $\neg\ fin\ mp$ **and** *no-confl*: $\neg\ inf\text{-}var\text{-}conflict\ (mp\text{-}mset\ mp)$
unfolding *p-inf-def* **using** *impr* **by** *fastforce*
from *inf*[*unfolded fin-def tvars-match-def*]
obtain $t\ l\ x$ **where** tl: $(t,l) \in\# mp$ **and** x: $x \in vars\ t$ **and** *inf*: *inf-sort* (*snd x*) **by** *auto*
from *get-var*[*OF - tl*] *mp tl* **obtain** y **where** ty: $(t,\ Var\ y) \in\# mp$ **by** *auto*
have $\exists\ x.\ (Var\ x,\ Var\ y) \in\# mp \wedge inf\text{-}sort\ (snd\ x)$
proof (*cases t*)
case (*Var z*)
with *ty inf x* **show** *?thesis* **by** (*intro exI*[*of - z*], *auto*)
next
case (*Fun f ts*)
from *fun-case*[*OF - ty*[*unfolded Fun*]] *mp* **show** *?thesis* **by** *auto*
qed
then obtain x **where** xy: $(Var\ x,\ Var\ y) \in\# mp$ **and** *inf*: *inf-sort* (*snd x*) **by** *auto*

```
                from mset-add[OF xy] obtain mp' where mp': mp = add-mset
(Var x, Var y) mp' by auto
              from nsvar[simplified, rule-format, OF mp[unfolded mp']] obtain s
y' where
                sy': (s,y') ∈# mp' and y': y ∈ vars y' by force
              from mset-add[OF sy'] mp' obtain mp'' where
                mp'': mp = add-mset (s,y') (add-mset (Var x, Var y) mp'')
                by auto
              from get-var[OF mp-mset-in-pat-mset[OF mp[unfolded mp']]] y'
                have mp'': mp = add-mset (s, Var y) (add-mset (Var x, Var y)
mp'')
                unfolding mp'' by (cases y', auto)
              from ndup mp'' mp have sx: s ≠ Var x by auto
               from no-clash mp'' mp have no-clash: ¬ Conflict-Clash s (Var x)
by metis
              from no-confl[unfolded inf-var-conflict-def not-ex, rule-format, of s
y Var x x] mp'' inf
              have ¬ Conflict-Var s (Var x) x by auto
             with sx no-clash have False by (cases s, auto simp: conflicts.simps
split: if-splits)
              thus ?thesis by auto
            qed
          qed auto
        qed
      next
        case no-non-inf: False
        have ∃ ps. p + {#} ⇒m ps
        proof (intro exI, rule pat-inf-var-conflict[OF - pne], intro 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 ≠ 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 (auto simp: tvars-pat-def)
        thus ?thesis by auto
      qed
    qed
  qed
```

qed
qed
qed

context
assumes *non-improved*: $\neg$ *improved*
begin

lemma *pat-fail-or-trans*: *wf-pat* (*pat-mset p*) $\Longrightarrow$ *pat-fail p* $\vee$ ($\exists$ *ps*. *p* $\Rightarrow_m$ *ps*)
using *pat-fail-or-trans-or-finite-var-form*[*of p*] *non-improved* **by** *auto*

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

theorem *P-step-NF*:
assumes *wf*: *wf-pats* (*pats-mset P*) **and** *NF*: *P* $\in$ *NF* $\Rrightarrow$
shows *P* $\in$ {{#}, *bottom-mset*}
proof (*rule ccontr*)
assume *nNF*: *P* $\notin$ {{#}, *bottom-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* = *add-mset p P'*
using *multiset-cases* **by** *auto*
with *wf* **have** *wf-pat* (*pat-mset p*) **by** (*auto simp*: *wf-pats-def*)
with *pat-fail-or-trans*
obtain *ps* **where** *pat-fail p* $\vee$ *p* $\Rightarrow_m$ *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

context
assumes *improved*: *improved*
and *inf*: *infinite* (*UNIV* :: *'v set*)
begin

lemma *pat-fail-or-trans-or-fvf*:
fixes *p* :: (*'f*,*'v*,*'s*) *pat-problem-mset*
assumes *wf-pat* (*pat-mset p*)
shows *pat-fail p* $\vee$ ($\exists$ *ps*. *p* $\Rightarrow_m$ *ps*) $\vee$ *finite-var-form-pat C* (*pat-mset p*)
using *assms pat-fail-or-trans-or-finite-var-form*[*of p*, *OF inf*] **by** *auto*

Normal forms only consist of finite-var-form pattern problems

theorem *P-step-NF-fvf*:
assumes *wf*: *wf-pats* (*pats-mset P*)
and *NF*: (*P*::(*'f*,*'v*,*'s*) *pats-problem-mset*) $\in$ *NF* $\Rrightarrow$
and *p*: *p* $\in$# *P*
shows *finite-var-form-pat C* (*pat-mset p*)
proof (*rule ccontr*)

assume *nfvf*: ¬ *?thesis*
from *wf p* **have** *wfp*: *wf-pat* (*pat-mset p*) **by** (*auto simp*: *wf-pats-def*)
from *mset-add*[*OF p*] **obtain** *P'* **where** *P*: *P* = *add-mset p P'* **by** *auto*
from *NF* **have** *NF*: ¬ (∃ *Q*. *P* ⇛$_m$ *Q*) **unfolding** *P-step-def* **by** *blast*
from *pat-fail-or-trans-or-fvf*[*OF wfp*] *nfvf*
obtain *ps* **where** *pat-fail p* ∨ *p* ⇒$_m$ *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*] *NF* **have** *P* = {# {#} #} **by** *auto*
with *P* **have** *p* = {#} **by** *auto*
with *nfvf* **show** *False* **unfolding** *finite-var-form-pat-def* **by** *auto*
qed

end

end

7.3 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*)*term* ⇒ ('*f*,'*w*)*term* ⇒ *nat* **where**
fun-diff l (*Var x*) = *num-funs l*
| *fun-diff* (*Fun g ls*) (*Fun f ts*) = (*if f* = *g* ∧ *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*: (⋀ *y*. *y* ∈# *N* ⟹ (*y*,*x*) ∈ *R*) ⟹ (*N* + *M*, *add-mset x M*) ∈ *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* ≤ *num-funs l*
proof (*induct l t rule*: *fun-diff.induct*)
case (*2 f ls g ts*)
show *?case*
proof (*cases f* = *g* ∧ *length ts* = *length ls*)
case *True*
have *sum-list* (*map2 fun-diff ls ts*) ≤ *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* · *σ*) ≤ *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* ⇒ *nat*) ⇒ '*a multiset* ⇒ *nat* **where**
sum-ms f ms ≡ *sum-mset* (*image-mset f ms*)

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

definition *max-size* :: '*s* ⇒ *nat* **where**
max-size s = (*if s* ∈ *S* ∧ ¬ *inf-sort s then Maximum* (*size* ' {*t*. *t* : *s in* $\mathcal{T}(C)$}) *else 0*)

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

definition *meas-symbols* :: ('*f*,'*v*,'*s*)*pat-problem-mset* ⇒ *nat* **where**
meas-symbols = *sum-ms* (*sum-ms* (λ (*t*,*l*). *num-funs t*))

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

definition *rel-pat* :: (('*f*,'*v*,'*s*)*pat-problem-mset* × ('*f*,'*v*,'*s*)*pat-problem-mset*)*set* (‹≺›) **where**
(≺) = *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** ‹≻› *50*) **where**
pp ≻ *pp'* ≡ (*pp'*,*pp*) ∈ ≺

definition *rel-pats* :: ((('*f*,'*v*,'*s*)*pats-problem-mset* × ('*f*,'*v*,'*s*)*pats-problem-mset*)*set* (‹≺*mul*›) **where**
≺*mul* = *mult* (≺)

abbreviation *gt-rel-pats* (**infix** ‹≻*mul*› *50*) **where**
P ≻*mul P'* ≡ (*P'*,*P*) ∈ ≺*mul*

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

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

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

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

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

lemma *meas-finvars-mono*: **assumes** *tvars-match* (*mp-mset mp*) $\subseteq$ *tvars-match* (*mp-mset mp'*)
shows *meas-finvars* {#*mp*#} $\leq$ *meas-finvars* {#*mp'*#}
using *tvars-match-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-match-mono*[*OF this*]]
show *?thesis* **unfolding** *meas-def rel-pat-def* **by** *auto*
qed

lemma *rel-mp-mp-step-mset*:
fixes *mp* :: ('*f*,'*v*,'*s*) *match-problem-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-match* (*mp-mset mp'*) $\subseteq$ *tvars-match* (*mp-mset mp*)
unfolding *tvars-match-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 *length ts* = *length ls* $\Longrightarrow$ ($\sum$ (*t*, *l*)∈#*mset* (*zip ts ls*). *num-funs t*) $\leq$ *sum-list* (*map num-funs ts*)
by (*induct ts ls rule*: *list-induct2*, *auto*)

```
  hence meas-symbols {#mp'#} < meas-symbols {#mp#}
    unfolding meas-def ∗ using ∗(3)
    by (auto simp: sum-mset-sum-list)
  ultimately show ?thesis unfolding rel-pat-def by auto
next
  case ∗: (match-decompose' mp1 y f n mp2 ys)
  let ?Var = Var :: 'v ⇒ ('f, 'v) term
  have meas-diff {#mp'#} ≤ meas-diff {#mp#}
   ⟷ (∑ (ti, yi)∈#(∑ (t, l)∈#mp1. mset (zip (args t) (map ?Var ys))). fun-diff
yi ti)
    ≤ (∑ (t, l)∈#mp1. fun-diff l t) (is - ⟷ ?sum ≤ -)
    unfolding ∗ meas-diff-def by simp
  also have ?sum = 0
    by (intro sum-mset.neutral ballI, auto simp: set-zip)
  finally have meas-diff {#mp'#} ≤ meas-diff {#mp#} by simp
  moreover
  have meas-finvars {#mp'#} ≤ meas-finvars {#mp#}
  proof (rule meas-finvars-mono)
    show tvars-match (mp-mset mp') ⊆ tvars-match (mp-mset mp)
      unfolding tvars-match-def ∗ using ∗(3,6)
      by (auto simp: set-zip set-conv-nth)
          (metis case-prod-conv nth-mem option.simps(3) root.elims term.sel(4)
term.set-intros(4))
  qed
  moreover
  have meas-symbols {#mp'#} < meas-symbols {#mp#}
  proof −
    from ‹2 ≤ size mp1› obtain T L MP where mp1: mp1 = add-mset (T,L)
MP
      by (cases mp1; force)
    from ∗(3)[of T L] mp1 obtain TS where id: T = Fun f TS L = Var y and
lTS: length TS = n
      by (cases T, auto)
    have aux: length ts = length ls ⟹
      (∑ (t, l)∈#mset (zip ts ls). num-funs t) ≤ sum-list (map num-funs ts)
      for ts :: ('f, nat × 's)term list and ls :: ('f,'v)term list
      by (induct ts ls rule: list-induct2, auto)
    have meas-symbols {#mp'#} < meas-symbols {#mp#} ⟷
    ((∑ (t, l)∈#mset (zip TS (map ?Var ys)). num-funs t) +
     (∑ (ti, yi)∈#(∑ (t, l)∈#MP. mset (zip (args t) (map ?Var ys))). num-funs
ti)
     ≤ (sum-list (map num-funs TS) + (∑ (t, l)∈#MP. num-funs t)))
      (is - ⟷ (?a + ?b ≤ ?c + ?d))
        unfolding meas-symbols-def ∗ mp1 id by (simp add: sum-mset-sum-list
less-Suc-eq-le)
    also have ...
    proof (rule add-le-mono)
      show ?a ≤ ?c using aux lTS ‹length ys = n› by auto
      from ∗(3) mp1 have (t, l) ∈# MP ⟹ l = Var y ∧ root t = Some (f, n)
```

for *l t* **by** *auto*
thus *?b* ≤ *?d*
proof (*induct MP*)
case (*add pair MP*)
obtain *t l* **where** *pair*: *pair* = (*t*,*l*) **by** *force*
from *add*(*2*)[*of t l*] **obtain** *ts* **where** *id*: *l* = *Var y t* = *Fun f ts* **and** *lts*: *length ts* = *n*
by (*cases t*, *auto simp*: *pair*)
from *add*(*1*)[*OF add*(*2*)]
have *IH*: (∑ (*ti*, *yi*)∈#(∑ (*t*, *l*)∈#*MP*. *mset* (*zip* (*args t*) (*map ?Var ys*))). *num-funs ti*)
≤ (∑ (*t*, *l*)∈#*MP*. *num-funs t*) **by** *auto*
from *IH aux*[*of ts*, *unfolded lts*, *of map ?Var ys*] ‹*length ys* = *n*›
show *?case* **unfolding** *pair id* **by** *auto*
qed *auto*
qed
finally show *meas-symbols* {#*mp*′#} < *meas-symbols* {#*mp*#} .
qed
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*) ≤ *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*′ ⊆# *p*
shows *meas-diff p*′ ≤ *meas-diff p*
meas-finvars p′ ≤ *meas-finvars p*
meas-symbols p′ ≤ *meas-symbols p*
proof −
from *sub* **obtain** *p*′′ **where** *p*: *p* = *p*′ + *p*′′ **by** (*metis subset-mset.less-eqE*)
show *meas-diff p*′ ≤ *meas-diff p meas-finvars p*′ ≤ *meas-finvars p meas-symbols p*′ ≤ *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*: $p = add\text{-}mset\ x\ p' + p''$
by (*metis multi-nonempty-split subset-mset.lessE union-mset-add-mset-left union-mset-add-mset-right*)
hence *lt*: $meas\text{-}setsize\ p' < meas\text{-}setsize\ p$ **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 $\exists\ t.\ t : s\ in\ \mathcal{T}(C) \wedge max\text{-}size\ s = size\ t \wedge (\forall\ t'.\ t' : s\ in\ \mathcal{T}(C) \longrightarrow size\ t' \leq size\ t)$
proof −
let *?set* = $\lambda\ s.\ size\ `\ \{t.\ t : s\ in\ \mathcal{T}(C)\}$
have *m*: *max-size s* = *Maximum* (*?set s*) **unfolding** *o-def max-size-def* **using** *inf sS* **by** *auto*
from *inf inf-sort-not-bdd*[*OF sS*] **have** *bdd-above* (*?set s*) **by** *auto*
moreover have *?set s* $\neq$ {} **by** (*auto intro*!: *sorts-non-empty sS*)
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*: $t : s\ in\ \mathcal{T}(C)$
shows $size\ t \leq max\text{-}size\ s$
using *max-size-term-of-sort*[*OF sS inf*] *sort* **by** *auto*

lemma *finite-sort-size*: **assumes** *c*: $c : map\ snd\ vs \to s\ in\ C$
and *inf*: $\neg$ *inf-sort s*
shows $sum\ (max\text{-}size\ o\ snd)\ (set\ vs) < max\text{-}size\ s$
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* $\exists\ v \in set\ vs.\ inf\text{-}sort\ (snd\ v)$)
case *True*
{
fix *v*
assume $v \in set\ vs$
with *vsS* **have** *v*: *snd v* $\in S$ **by** *auto*
note *sorts-non-empty*[*OF this*]
}

hence $\forall\ v.\ \exists\ t.\ v \in set\ vs \longrightarrow t : snd\ v\ in\ \mathcal{T}(C)$ **by** *auto*
from *choice*[*OF this*] **obtain** t **where**
t: $\bigwedge v.\ v \in set\ vs \Longrightarrow t\ v : snd\ v\ in\ \mathcal{T}(C)$ **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*
note *nbdd* = *inf-sort-not-bdd*[*OF vlS*, *THEN iffD2*, *OF inf-vl*]
from *not-bdd-above-natD*[*OF nbdd*, *of ?m*] *t*[*OF vl*]
obtain *tl* **where**
tl: $tl : snd\ vl\ in\ \mathcal{T}(C)$ **and** *large*: $?m \leq size\ tl$ **by** *fastforce*
let $?t = Fun\ c\ (map\ (\lambda\ v.\ if\ v = vl\ then\ tl\ else\ t\ v)\ vs)$
have $?t : s\ in\ \mathcal{T}(C)$
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*: $\neg$ *inf-sort* (*snd v*) **by** *auto*
from *vsS v* **have** $snd\ v \in S$ **by** *auto*
from *max-size-term-of-sort*[*OF this inf*]
have $\exists\ t.\ t : snd\ v\ in\ \mathcal{T}(C) \wedge size\ t = max\text{-}size\ (snd\ v)$ **by** *auto*
}
hence $\forall\ v.\ \exists\ t.\ v \in set\ vs \longrightarrow t : snd\ v\ in\ \mathcal{T}(C) \wedge size\ t = max\text{-}size\ (snd\ v)$ **by** *auto*
from *choice*[*OF this*] **obtain** t **where**
t: $v \in set\ vs \Longrightarrow t\ v : snd\ v\ in\ \mathcal{T}(C) \wedge size\ (t\ v) = max\text{-}size\ (snd\ v)$ **for** v **by** *blast*
let $?t = Fun\ c\ (map\ t\ vs)$
have $?t : s\ in\ \mathcal{T}(C)$
by (*intro Fun-hastypeI*[*OF c*] *list-all2-map-map*, *insert t*, *auto*)
from *max-size-max*[*OF sS inf this*]
have $size\ ?t \leq max\text{-}size\ s$ **.**

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

lemma *rel-pp-step-mset*:
fixes p :: $('f,'v,'s)$ *pat-problem-mset*

assumes $p \Rightarrow_m ps$
and $p' \in\# ps$
shows $p \succ p'$
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** $\exists$ *s t*. (*s*,*t*) $\in\#$ *mp* $\wedge$ (*s* = *Var x* $\wedge$ *is-Fun t*
$\vee$ (*x* $\in$ *vars s* $\wedge$ $\neg$ *inf-sort* (*snd x*)))
proof
assume $*$: (*s*, *Var y*) $\in\#$ *mp* $\wedge$ (*t*, *Var y*) $\in\#$ *mp* $\wedge$ *Conflict-Var s t x* $\wedge$ $\neg$ *inf-sort* (*snd x*)
$\wedge$ (*improved* $\longrightarrow$ *s* = *Var x* $\wedge$ *is-Fun t*)
hence *Conflict-Var s t x* **and** $\neg$ *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*) $\in\#$ *mp* **and** *choice*: *s* = *Var x* $\wedge$ *is-Fun t* $\vee$ *x* $\in$ *vars s* $\wedge$ $\neg$ *inf-sort* (*snd x*)
by *auto*
let *?p* = *add-mset mp p*
let *?s* = *snd x*
from $*(3)$ τ*s-list*
obtain τ **where** τ: $\tau \in \tau s\ n\ x$ **and** *p'*: $p' =$ *subst-pat-problem-mset* τ *?p* **by** *auto*

let *?tau-mset* = *subst-pat-problem-mset* τ :: ('*f*,'*v*,'*s*) *pat-problem-mset* $\Rightarrow$ -
let *?tau* = *subst-match-problem-mset* τ :: ('*f*,'*v*,'*s*) *match-problem-mset* $\Rightarrow$ -
from τ[*unfolded* τ*s-def* τ*c-def List.maps-def*]
obtain *c sorts* **where** *c*: *c* : *sorts* $\rightarrow$ *?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* $\in$ *S* **and** *sorts*: *set sorts* $\subseteq$ *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*) $\subseteq$ *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*) $\in$ *vars* (τ *y*) $\Longrightarrow$ *b* $\in$ *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* $\succ$ *?tau-mset ?p*

proof
assume $s = Var\ x \wedge is\text{-}Fun\ t$
then obtain $f\ ts$ **where** s: $s = Var\ x$ **and** t: $t = Fun\ f\ ts$ **by** $(cases\ t,\ auto)$
have $meas\text{-}diff\ (?tau\text{-}mset\ ?p) =$
$meas\text{-}diff\ (?tau\text{-}mset\ (add\text{-}mset\ mp'\ p)) + fun\text{-}diff\ t\ (s \cdot \tau)$
unfolding $meas\text{-}def\ subst\text{-}defs\ subst\text{-}left\text{-}def\ mp$ **by** $simp$
also have $\ldots \leq meas\text{-}diff\ (add\text{-}mset\ mp'\ p) + fun\text{-}diff\ t\ (\tau\ x)$ **using** $meas\text{-}diff\text{-}subst\text{-}le[of\ \tau]\ s$ **by** $auto$
also have $\ldots < meas\text{-}diff\ (add\text{-}mset\ mp'\ p) + fun\text{-}diff\ t\ s$
proof $(rule\ add\text{-}strict\text{-}left\text{-}mono)$
have $fun\text{-}diff\ t\ (\tau\ x) < num\text{-}funs\ t$
unfolding $tau\ subst\text{-}simps\ fun\text{-}diff.simps$
by $(rule\ fun\text{-}diff\text{-}num\text{-}funs\text{-}lt[OF\ refl],\ auto\ simp\text{:}\ t)$
thus $fun\text{-}diff\ t\ (\tau\ x) < fun\text{-}diff\ t\ s$ **by** $(auto\ simp\text{:}\ s\ t)$
qed
also have $\ldots = meas\text{-}diff\ ?p$ **unfolding** $mp\ meas\text{-}def$ **by** $auto$
finally show $?thesis$ **unfolding** $rel\text{-}pat\text{-}def$ **by** $auto$
next
assume $x \in vars\ s \wedge \neg\ inf\text{-}sort\ (snd\ x)$
hence x: $x \in vars\ s$ **and** inf: $\neg\ inf\text{-}sort\ (snd\ x)$ **by** $auto$
from $meas\text{-}diff\text{-}subst\text{-}le[of\ \tau]$
have fd: $meas\text{-}diff\ p' \leq meas\text{-}diff\ ?p$ **unfolding** p' **.**
have $meas\text{-}finvars\ (?tau\text{-}mset\ ?p) = meas\text{-}finvars\ (?tau\text{-}mset\ \{\#mp\#\}) + meas\text{-}finvars\ (?tau\text{-}mset\ p)$
unfolding $subst\text{-}defs\ meas\text{-}def$ **by** $auto$
also have $\ldots < meas\text{-}finvars\ \{\#mp\#\} + meas\text{-}finvars\ p$
proof $(rule\ add\text{-}less\text{-}le\text{-}mono)$
have $vars\text{-}\tau\text{-}var$: $vars\ (\tau\ y) = (if\ x = y\ then\ set\ vs\ else\ \{y\})$ **for** y **unfolding** $\tau\ subst\text{-}def$ **by** $auto$
have $vars\text{-}\tau$: $vars\ (t \cdot \tau) = vars\ t - \{x\} \cup (if\ x \in vars\ t\ then\ set\ vs\ else\ \{\})$ **for** t
unfolding $vars\text{-}term\text{-}subst\ image\text{-}comp\ o\text{-}def\ vars\text{-}\tau\text{-}var$ **by** $auto$
have $tvars\text{-}match\text{-}subst$: $tvars\text{-}match\ (mp\text{-}mset\ (?tau\ mp)) =$
$tvars\text{-}match\ (mp\text{-}mset\ mp) - \{x\} \cup (if\ x \in tvars\text{-}match\ (mp\text{-}mset\ mp)\ then\ set\ vs\ else\ \{\})$ **for** mp
unfolding $subst\text{-}defs\ subst\text{-}left\text{-}def\ tvars\text{-}match\text{-}def$
by $(auto\ simp\text{:}vars\text{-}\tau\ split\text{:}\ if\text{-}splits\ prod.split)$
have $id1$: $meas\text{-}finvars\ (?tau\text{-}mset\ \{\#mp\#\}) = (\sum x \in tvars\text{-}match\ (mp\text{-}mset\ (?tau\ mp)).\ max\text{-}size\ (snd\ x))$ **for** mp
unfolding $meas\text{-}def\ subst\text{-}defs$ **by** $auto$
have $id2$: $meas\text{-}finvars\ \{\#mp\#\} = (\sum x \in tvars\text{-}match\ (mp\text{-}mset\ mp).\ max\text{-}size\ (snd\ x))$
for $mp :: ('f,'v,'s)\ match\text{-}problem\text{-}mset$
unfolding $meas\text{-}def\ subst\text{-}defs$ **by** $simp$
have eq: $x \notin tvars\text{-}match\ (mp\text{-}mset\ mp) \Longrightarrow meas\text{-}finvars\ (?tau\text{-}mset\ \{\#\ mp\ \#\}) = meas\text{-}finvars\ \{\#mp\#\}$ **for** mp
unfolding $id1\ id2$ **by** $(rule\ sum.cong[OF\ -\ refl],\ auto\ simp\text{:}\ tvars\text{-}match\text{-}subst)$
{
fix $mp :: ('f,'v,'s)\ match\text{-}problem\text{-}mset$

assume *xmp*: $x \in$ *tvars-match* (*mp-mset mp*)
let *?mp* = (*mp-mset mp*)
have *fin*: *finite* (*tvars-match ?mp*) **by** (*rule tvars-match-fin*)
define *Mp* **where** *Mp* = *tvars-match ?mp* − {*x*}
from *xmp* **have** *1*: *tvars-match* (*mp-mset* (*?tau mp*)) = *set vs* ∪ *Mp*
unfolding *tvars-match-subst Mp-def* **by** *auto*
from *xmp* **have** *2*: *tvars-match ?mp* = *insert x Mp* **and** *xMp*: $x \notin$ *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* #}) ≤ *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-match-def*)

show *meas-finvars* (*?tau-mset p*) ≤ *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*)

```
next
  case ∗: (pat-inf-var-conflict p)
  thus ?case by (intro meas-sub-rel-pat, cases p, auto)
qed
```

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

```
lemma rel-P-trans:
  assumes P ⇛_m P'
  shows P ≻mul P'
  using assms
proof induct
  case ∗: (P-failure p P)
  from ∗ have p ≠ {#} ∨ p = {#} ∧ P ≠ {#} by auto
  thus ?case
  proof
    assume p ≠ {#}
    then obtain mp p' where p: p = add-mset mp p' by (cases p, auto)
    have p ≻ {#} 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 = {#} ∧ P ≠ {#} 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 ⇛
proof −
  have sub: ⇛ ⊆ ≺mul^−1
    using rel-P-trans unfolding P-step-def by auto
  show ?thesis
    apply (rule SN-subset[OF - sub])
    apply (rule wf-imp-SN)
    using wf-rel-pats by simp
qed
```

7.4 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*)
next
  case $*$: (*match-decompose' mp y f n mp' ys*)
  show *?case* **by** (*rule mp-step-cong*[*OF mp-decompose'*[*OF* $*$(*1*,*2*) $*$(*3*)[*unfolded set-mset-union*] $*$(*4*,*6*)]], *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*)
next
  case $*$: (*match-clash-sort s t x mp*)
  show *?case* **by** (*rule mp-fail-cong*[*OF mp-clash-sort*], *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-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 ∈ set-mset P}
 from ∗(1)
 have insert (pat-mset p) (pats-mset P) ⇛s pats-mset ps ∪ 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 ∈ tvars-match (mp-mset mp)
     using conflicts(4)[of s t x] unfolding tvars-match-def
     by (auto intro!:term.set-intros(3))
   hence x: x ∈ tvars-pat (pat-mset (add-mset mp p)) unfolding tvars-pat-def
     using ∗(2) by auto
   show ?case unfolding conv τs-list
     apply (rule P-step-set-cong[OF P-instantiate[OF ∗(1) x]])
     by (unfold conv subst-defs set-map image-comp, auto)
 next
   case ∗: (pat-inf-var-conflict pp pp′)
   from pp-inf-var-conflict[OF ∗(1), of pat-mset pp′]
   have pat-mset (pp + pp′) ⇒s pat-mset pp′
     using ∗ by (auto simp: tvars-pat-def image-Un)
   from P-simp[OF this]
   show ?case by auto
 qed
 thus ?case unfolding conv .
qed

lemma P-step-pp-trans: assumes (P,Q) ∈ ⇛
 shows pats-mset P ⇛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) ∈
P-step
shows wf-pats (pats-mset Q) ∧ (pats-complete C (pats-mset P) = pats-complete C
```

(*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*) ∈ ⇛*
shows *wf-pats* (*pats-mset Q*) ∧ (*pats-complete C* (*pats-mset P*) ⟷ *pats-complete C* (*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 *non-improved*: ¬ *improved*
and *wf*: *wf-pats* (*pats-mset P*) **and** *NF*: (*P*,*Q*) ∈ ⇛!
shows *Q* = {#} ∧ *pats-complete C* (*pats-mset P*) — either the result is and input P is complete
∨ *Q* = *bottom-mset* ∧ ¬ *pats-complete C* (*pats-mset P*) — or the result = bot and P is not complete
proof −
from *NF* **have** *steps*: (*P*,*Q*) ∈ ⇛^* **and** *NF*: *Q* ∈ *NF P-step* **by** *auto*
from *P-steps-pcorrect*[*OF wf steps*]
have *wf*: *wf-pats* (*pats-mset Q*) **and**
sound: *pats-complete C* (*pats-mset P*) = *pats-complete C* (*pats-mset Q*)
by *blast*+
from *P-step-NF*[*OF non-improved wf NF*] **have** *Q* ∈ {{#},*bottom-mset*} **.**
thus *?thesis* **unfolding** *sound* **by** *auto*
qed

theorem *P-step-improved*:
fixes *P* :: (′*f*,′*v*,′*s*) *pats-problem-mset*
assumes *improved*
and *inf*: *infinite* (*UNIV* :: ′*v set*)
and *wf*: *wf-pats* (*pats-mset P*) **and** *NF*: (*P*,*Q*) ∈ ⇛!
shows *pats-complete C* (*pats-mset P*) ⟷ *pats-complete C* (*pats-mset Q*) — equivalence
p ∈# *Q* ⟹ *finite-var-form-pat C* (*pat-mset p*) — all remaining problems are in finite-var-form
proof −
from *NF* **have** *steps*: (*P*,*Q*) ∈ ⇛^* **and** *NF*: *Q* ∈ *NF P-step* **by** *auto*
note ∗ = *P-steps-pcorrect*[*OF wf steps*]
from ∗
show *pats-complete C* (*pats-mset P*) = *pats-complete C* (*pats-mset Q*) **..**
from ∗ **have** *wfQ*: *wf-pats* (*pats-mset Q*) **by** *auto*
from *P-step-NF-fvf*[*OF* ‹*improved*› *inf this NF*]
show *p* ∈# *Q* ⟹ *finite-var-form-pat C* (*pat-mset p*) **.**

qed

end
end

8 A List-Based Implementation to Decide Pattern Completeness

theory *Pattern-Completeness-List*
imports
Pattern-Completeness-Multiset
Compute-Nonempty-Infinite-Sorts
Finite-IDL-Solver-Interface
HOL−Library.AList
HOL−Library.Mapping
Singleton-List
begin

8.1 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 $('f,'v,'s)match\text{-}problem\text{-}fvf = ('v,(nat \times 's)\ list)\ alist$
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 $('f,'v,'s)pat\text{-}problem\text{-}lx = ('f,'v,'s)match\text{-}problem\text{-}lx\ list$
type-synonym $('f,'v,'s)pat\text{-}problem\text{-}fvf = ('f,'v,'s)match\text{-}problem\text{-}fvf\ list$
type-synonym $('f,'v,'s)pats\text{-}problem\text{-}list = ('f,'v,'s)pat\text{-}problem\text{-}list\ list$
type-synonym $('f,'v,'s)pat\text{-}problem\text{-}set\text{-}impl = (('f,nat \times 's)term \times ('f,'v)term)\ list\ list$

definition $lvars\text{-}mp :: ('f,'v,'s)match\text{-}problem\text{-}mset \Rightarrow 'v\ set$ **where**
$lvars\text{-}mp\ mp = (\bigcup\ (vars\ `\ snd\ `\ mp\text{-}mset\ mp))$

definition $vars\text{-}mp\text{-}mset :: ('f,'v,'s)match\text{-}problem\text{-}mset \Rightarrow 'v\ multiset$ **where**
$vars\text{-}mp\text{-}mset\ mp = sum\text{-}mset\ (image\text{-}mset\ (vars\text{-}term\text{-}ms\ o\ snd)\ mp)$

definition *ll-mp* :: (*'f*,*'v*,*'s*)*match-problem-mset* ⇒ *bool* **where**
ll-mp mp = (∀ *x*. *count* (*vars-mp-mset mp*) *x* ≤ *1*)

definition *ll-pp* :: (*'f*,*'v*,*'s*)*pat-problem-list* ⇒ *bool* **where**
ll-pp p = (∀ *mp* ∈ *set p*. *ll-mp* (*mset mp*))

definition *lvars-pp* :: (*'f*,*'v*,*'s*)*pat-problem-mset* ⇒ *'v set* **where**
lvars-pp pp = (⋃ (*lvars-mp* ‘ *set-mset pp*))

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

where *mp-list* ≡ *mset*

abbreviation *mp-lx* :: (*'f*,*'v*,*'s*)*match-problem-lx* ⇒ (*'f*,*'v*,*'s*)*match-problem-list*
where *mp-lx* ≡ *map* (*map-prod Var id*)

definition *mp-rx* :: (*'f*,*'v*,*'s*)*match-problem-rx* ⇒ (*'f*,*'v*,*'s*)*match-problem-mset*
where *mp-rx mp* = *mset* (*List.maps* (λ (*x*,*ts*). *map* (λ *t*. (*t*,*Var x*)) *ts*) (*fst mp*))

definition *mp-rx-list* :: (*'f*,*'v*,*'s*)*match-problem-rx* ⇒ (*'f*,*'v*,*'s*)*match-problem-list*
where *mp-rx-list mp* = *List.maps* (λ (*x*,*ts*). *map* (λ *t*. (*t*,*Var x*)) *ts*) (*fst mp*)

definition *mp-lr* :: (*'f*,*'v*,*'s*)*match-problem-lr* ⇒ (*'f*,*'v*,*'s*)*match-problem-mset*
where *mp-lr pair* = (*case pair of* (*lx*,*rx*) ⇒ *mp-list* (*mp-lx lx*) + *mp-rx rx*)

definition *mp-lr-list* :: (*'f*,*'v*,*'s*)*match-problem-lr* ⇒ (*'f*,*'v*,*'s*)*match-problem-list*
where *mp-lr-list pair* = (*case pair of* (*lx*,*rx*) ⇒ *mp-lx lx* @ *mp-rx-list rx*)

definition *pat-lr* :: (*'f*,*'v*,*'s*)*pat-problem-lr* ⇒ (*'f*,*'v*,*'s*)*pat-problem-mset*
where *pat-lr ps* = *mset* (*map mp-lr ps*)

definition *pat-lx* :: (*'f*,*'v*,*'s*)*pat-problem-lx* ⇒ (*'f*,*'v*,*'s*)*pat-problem-mset*
where *pat-lx ps* = *mset* (*map* (*mp-list o mp-lx*) *ps*)

definition *pat-mset-list* :: (*'f*,*'v*,*'s*)*pat-problem-list* ⇒ (*'f*,*'v*,*'s*)*pat-problem-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* ⇒ *'v list* × (*'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 (*xs*,(*xl*,(*filter* (λ (*x*,*ts*). *tl ts* ≠ [] ∨ *x* ∈ *set xs*) *rx*),*b*)))

definition *find-var* :: *bool* ⇒ (*'f*,*'v*,*'s*)*match-problem-lr list* ⇒ - **where**
find-var improved p = (*case List.maps* (λ (*lx*,-). *lx*) *p of*
(*x*,*t*) # - ⇒ *Some x*
| [] ⇒ *if improved then* (*let flat-mps* = *List.maps* (*fst o snd*) *p in*
(*map-option* (λ (*x*,*ts*). *case find is-Var ts of Some* (*Var x*) ⇒ *x*)
(*find* (λ *rx*. ∃ *t* ∈ *set* (*snd rx*). *is-Fun t*) *flat-mps*)))
else Some (*let* (-,*rx*,*b*) = *hd 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* = [])

fun *zipAll* :: *'a list* ⇒ *'b list list* ⇒ (*'a* × *'b list*) *list* **where**
zipAll [] - = []
| *zipAll* (*x* # *xs*) *yss* = (*x*, *map hd yss*) # *zipAll xs* (*map tl yss*)

datatype (*'f*,*'v*,*'s*)*pat-impl-result* = *Incomplete*
| *New-Problems nat* × *nat* × (*'f*,*'v*,*'s*)*pat-problem-list list*
| *Fin-Var-Form* (*'f*,*'v*,*'s*)*pat-problem-fvf*

Transforming finite variable forms:

definition *tvars-match-list* = *remdups* ∘ *concat* ∘ *map* (*var-list-term* ∘ *fst*)

definition *tvars-pat-list* = *remdups* ∘ *concat* ∘ *map tvars-match-list*

definition *var-form-of-match-rx* :: (*'f*,*'v*,*'s*)*match-problem-rx* ⇒ (*'v* × (*nat* × *'s*)
list) *list* **where**
var-form-of-match-rx = *map* (*map-prod id* (*map the-Var*)) *o fst*

definition *match-of-var-form-list* **where**
match-of-var-form-list mpv = *concat* [[(*Var v*, *Var x*). *v* ← *vs*]. (*x*,*vs*) ← *mpv*]

definition *var-form-of-pat-rx* **where**
var-form-of-pat-rx = *map var-form-of-match-rx*

definition *pat-of-var-form-list* **where**
pat-of-var-form-list = *map match-of-var-form-list*

```
lemma size-zip[termination-simp]: length ts = length ls ⟹ size-list (λp. size (snd
p)) (zip ts ls)
  < Suc (size-list size ls)
  by (induct ts ls rule: list-induct2, auto)

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

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

definition bounds-list bnd cnf = (let vars = remdups (concat (concat cnf))
  in map (λ v. (v, int (bnd v) − 1)) vars)

fun pairs-of-list where
  pairs-of-list (x # y # xs) = (x,y) # pairs-of-list (y # xs)
| pairs-of-list - = []

lemma set-pairs-of-list: set (pairs-of-list xs) = { (xs ! i, xs ! (Suc i)) | i. Suc i <
length xs}
proof (induct xs rule: pairs-of-list.induct)
  case (1 x y xs)
  define n where n = length xs
  have id: {f i |i. Suc i < length (x # y # xs)}
    = insert (f 0) {f (Suc i) |i. Suc i < length (y # xs)} for f :: nat ⇒ 'a × 'a
    unfolding list.size n-def[symmetric]
    apply auto
    subgoal for a b i by (cases i, auto)
    done
  show ?case unfolding pairs-of-list.simps set-simps 1 id by auto
qed auto

lemma diff-pairs-of-list: (∃ x ∈ set xs. ∃ y ∈ set xs. f x ≠ f y) ⟷
  (∃ (x,y) ∈ set (pairs-of-list xs). f x ≠ f y) (is ?l = ?r)
proof
  assume ?r
  from this[unfolded set-pairs-of-list] obtain i where i: Suc i < length xs
```

```
    and diff: f (xs ! i) ≠ f (xs ! (Suc i)) by auto
  from i have xs ! i ∈ set xs xs ! (Suc i) ∈ set xs by auto
  with diff show ?l by auto
next
  assume ?l
  show ?r
  proof (rule ccontr)
    let ?n = length xs
    assume ¬ ?r
    hence eq: ⋀ i. Suc i < ?n ⟹ f (xs ! i) = f (xs ! (Suc i)) by (auto simp:
set-pairs-of-list)
    have eq: i < ?n ⟹ f (xs ! i) = f (xs ! 0) for i
      by (induct i, insert eq, auto)
    hence ⋀ i j. i < ?n ⟹ j < ?n ⟹ f (xs ! i) = f (xs ! j) by auto
    with ‹?l› show False unfolding set-conv-nth by auto
  qed
qed

definition dist-pairs-list cnf = map (List.maps pairs-of-list) cnf

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))))
      )))
```

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 pat-lin-impl :: nat ⇒ ('f,'v,'s)pat-problem-list ⇒ ('f,'v,'s)pat-problem-list
list option where
  pat-lin-impl n p = (case pat-inner-lin-impl p [] of None ⇒ Some []
     | Some p' ⇒ if p' = [] then None
       else (let x = fst (hd (hd p')); p'l = map mp-lx p' in
         Some (map (λ τ. subst-pat-problem-list τ p'l) (τs-list n x))))

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

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

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

context
  fixes
  CC :: 'f × 's list ⇒ 's option and
  renNat :: nat ⇒ 'v and
  renVar :: 'v ⇒ 'v and
  fidl-solver :: ((nat × 's) × int) list × ((nat × 's) × (nat × 's)) list list ⇒ bool
begin

partial-function (tailrec) decomp'-main-loop where
  decomp'-main-loop n xs list out = (case list of
     [] ⇒ (n, out) — one might change to (rev out) in order to preserve the order
   | ((x,ts) # rxs) ⇒ (if tl ts = [] ∨ (∃ t ∈ set ts. is-Var t) ∨ x ∈ set xs
    then decomp'-main-loop n xs rxs ((x,ts) # out)
    else let l = length (args (hd ts));
           fresh = map renNat [n ..< n + l];
           new = zipAll fresh (map args ts);
           cleaned = filter (λ (y,ts'). tl ts' ≠ []) (map (λ (y,ts'). (y, remdups ts'))
new)
       in decomp'-main-loop (n + l) xs (cleaned @ rxs) out))

definition decomp'-impl where
```

```
  decomp'-impl n xs mp = (case mp of
    (xl,(rx,b)) ⇒ case decomp'-main-loop n xs rx [] of
    (n', rx') ⇒ (n', (xl,(rx',b))))


definition apply-decompose' :: ('f,'v,'s)match-problem-lr ⇒ bool
  where apply-decompose' mp = (improved ∧ (case mp of (xl,(rx,b)) ⇒ (¬ b ∧ xl
= [])))

definition match-decomp'-impl :: nat ⇒ ('f,'v,'s)match-problem-list ⇒ (nat ×
('f,'v,'s)match-problem-lr) option where
  match-decomp'-impl n mp = map-option (λ (xs,mp).
    if apply-decompose' mp
    then decomp'-impl n xs mp else (n, mp)) (match-steps-impl mp)

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

definition pat-impl :: nat ⇒ nat ⇒ ('f,'v,'s)pat-problem-list ⇒ ('f,'v,'s)pat-impl-result
where
  pat-impl n nl p = (case pat-inner-impl nl p [] of None ⇒ New-Problems (n,nl,[])
      | Some (nl',p') ⇒ (case partition (λ mp. snd (snd mp)) p' of
        (ivc,no-ivc) ⇒ if no-ivc = [] then Incomplete — detected inf-var-conflict (or
empty mp)
        else (if improved ∧ ivc ≠ [] ∧ (∀ mp ∈ set no-ivc. fst mp = []) then
          New-Problems (n, nl', [map mp-lr-list (filter — inf-var-conflict' + match-
clash-sort
           ( λ mp. ∀ xts ∈ set (fst (snd mp)). is-singleton-list (map (T(CC,V)) (snd
xts))) no-ivc)])
        else (case find-var improved no-ivc of Some x ⇒ let p'l = map mp-lr-list p'


         in
          New-Problems (n + m, nl', map (λ τ. subst-pat-problem-list τ p'l) (τs-list
n x))
        | None ⇒ Fin-Var-Form (map (map (map-prod id (map the-Var)) o fst o
snd) no-ivc)))))

partial-function (tailrec) pats-impl :: nat ⇒ nat ⇒ ('f,'v,'s)pats-problem-list ⇒
bool where
  pats-impl n nl ps = (case ps of [] ⇒ True
    | p # ps1 ⇒ (case pat-impl n nl p of
        Incomplete ⇒ False
      | Fin-Var-Form p' ⇒
        let bnd = (cd-sort ∘ snd); cnf = (map (map snd) p')
```

```
      in if fidl-solver (bounds-list bnd cnf, dist-pairs-list cnf) then False else
        pats-impl n nl ps1
    | New-Problems (n',nl',ps2) ⇒ pats-impl n' nl' (ps2 @ ps1)))

definition pat-complete-impl :: ('f,'v,'s)pats-problem-list ⇒ bool where
  pat-complete-impl ps = (let
      n = Suc (max-list (List.maps (map fst o vars-term-list o fst) (concat (concat
ps))));
      nl = 0;
      ps' = if improved then map (map (map (apsnd (map-vars renVar)))) ps else
ps
    in pats-impl n nl ps')
end
end

definition renaming-funs :: (nat ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ bool where
  renaming-funs rn rx = (inj rn ∧ inj rx ∧ range rn ∩ range rx = {})

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.τs-list-def
  pattern-completeness-context.apply-decompose'-def
  pattern-completeness-context.decomp'-main-loop.simps
  pattern-completeness-context.decomp'-impl-def
  pattern-completeness-context.insert-rx-def
  pattern-completeness-context.decomp-impl.simps
  pattern-completeness-context.match-decomp'-impl-def
  pattern-completeness-context.match-steps-impl-def
  pattern-completeness-context.pat-inner-impl.simps
  pattern-completeness-context.pat-lin-impl-def
  pattern-completeness-context.pats-lin-impl.simps
  pattern-completeness-context.pat-complete-lin-impl-def

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

8.2 Partial Correctness of the Implementation

TODO: move

```
lemma mset-sum-reindex: (∑ x∈#A. image-mset (f x) B) = (∑ i∈#B. {#f x i.
x ∈# A#})
proof (induct A)
  case (add x A)
  show ?case
    by (simp add: add)
   (smt (verit, del-insts) add.commute add-mset-add-single image-mset-cong sum-mset.distrib
       sum-mset-singleton-mset)
qed auto
```

lemma *vars-mp-mset-add*: *vars-mp-mset* (*mp* + *mp'*) = *vars-mp-mset mp* + *vars-mp-mset mp'*
unfolding *vars-mp-mset-def* **by** *auto*

zipAll

lemma *zipAll*: **assumes** *length as* = *n*
and ⋀ *bs*. *bs* ∈ *set bss* ⟹ *length bs* = *n*
shows *zipAll as bss* = *map* (λ *i*. (*as* ! *i*, *map* (λ *bs*. *bs* ! *i*) *bss*)) [*0*..<*n*]
using *assms*
proof (*induct as arbitrary*: *n bss*)
case (*Cons a as sn bss*)
then obtain *n* **where** *sn*: *sn* = *Suc n* **by** *auto*
let *?tbss* = *map tl bss*
from *Cons*(*2*−) *sn* **have** *prems*: *length as* = *n* ⋀ *bs*. *bs* ∈ *set ?tbss* ⟹ *length bs* = *n*
by *auto*
from *Cons*(*2*−) *sn* **have** *hd*: *bs* ∈ *set bss* ⟹ *hd bs* = *bs* ! *0* **for** *bs* **by** (*cases bs*) *force*+
from *Cons*(*2*−) *sn* **have** *tl*: *bs* ∈ *set bss* ⟹ *tl bs* ! *i* = *bs* ! *Suc i* **for** *bs i* **by** (*cases bs*) *force*+
note *IH* = *Cons*(*1*)[*OF prems*, *of ?tbss*]
have *id*: [*0*..<*sn*] = *0* # *map Suc* [*0*..<*n*] **unfolding** *sn upt-0-Suc-Cons* **..**
show *?case* **unfolding** *id zipAll.simps list.simps map-map o-def*
by (*subst IH*, *insert hd tl*, *auto*)
qed *simp*

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

lemma *mset-concat-union*:
mset (*concat xs*) = ∑# (*mset* (*map mset xs*))
by (*induct xs*, *auto simp*: *union-commute*)

lemma *in-map-mset*[*intro*]:
a ∈# *A* ⟹ *f a* ∈# *image-mset f A*
unfolding *in-image-mset* **by** *simp*

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

lemma *mp-rx-append*: *mp-rx* (*xs* @ *ys*, *b*) = *mp-rx* (*xs*,*b*) + *mp-rx* (*ys*,*b*)
unfolding *mp-rx-def List.maps-def* **by** *auto*

lemma *mp-rx-Cons*: *mp-rx* (*p* # *xs*, *b*) = *mp-list* (*case p of* (*x*, *ts*) ⇒ *map* (λ*t*. (*t*, *Var x*)) *ts*)
+ *mp-rx* (*xs*,*b*)
unfolding *mp-rx-def List.maps-def* **by** *auto*

lemma *set-tvars-match-list*: *set* (*tvars-match-list mp*) = *tvars-match* (*set mp*)
by (*auto simp*: *tvars-match-list-def tvars-match-def*)

lemma *set-tvars-pat-list*: *set* (*tvars-pat-list pp*) = *tvars-pat* (*pat-list pp*)
by (*simp add*: *tvars-pat-list-def tvars-pat-def set-tvars-match-list pat-list-def*)

lemma *finite-var-form-pat-pat-complete-list*:
fixes *pp*::('*f*,'*v*,'*s*) *pat-problem-list* **and** *C*
assumes *fvf*: *finite-var-form-pat C* (*pat-list pp*)
and *pp*: *pp* = *pat-of-var-form-list fvf*
and *dist*: *Ball* (*set fvf*) (*distinct o map fst*)
shows *pat-complete C* (*pat-list pp*) ⟷
(∀ α. (∀ *v* ∈ *set* (*tvars-pat-list pp*). α *v* < *card-of-sort C* (*snd v*)) ⟶
(∃ *c* ∈ *set* (*map* (*map snd*) *fvf*).
∀ *vs* ∈ *set c*. *UNIQ* (α ' *set vs*)))
proof−
from *finite-var-form-imp-of-var-form-pat*[*OF fvf*]
have *vf*: *var-form-pat* (*pat-list pp*)**.**
have (∃ *mp* ∈ *pat-list pp*. ∀ *x*. *UNIQ* {α *v* |*v*. (*Var v*, *Var x*) ∈ *mp*}) ⟷
(∃ *mpv* ∈ *set fvf*. ∀ (*x*,*vs*) ∈ *set mpv*. *UNIQ* (α ' *set vs*))
(**is** *?l* ⟷ *?r*)
for α :: - ⇒ *nat*
proof *safe*
fix *mpv*
assume *mpv* ∈ *set fvf*
and *r*: ∀ (*x*,*vs*) ∈ *set mpv*. *UNIQ* (α ' *set vs*)
with *pp*[*unfolded pat-of-var-form-list-def*] *dist*
have *mem*: *set* (*match-of-var-form-list mpv*) ∈ *pat-list pp*
and *dist*: *distinct* (*map fst mpv*)
unfolding *pat-list-def* **by** *auto*
show *?l*
proof (*intro bexI*[*OF - mem*] *allI*)
fix *x*
show *UNIQ* {α *v* |*v*. (*Var v*, *Var x*) ∈ *set* (*match-of-var-form-list mpv*)} (**is** *UNIQ ?vs*)
proof (*cases x* ∈ *fst* ' *set mpv*)
case *False*
hence *vs*: *?vs* = {} **unfolding** *match-of-var-form-list-def* **by** *force*
show *?thesis* **unfolding** *vs* **using** *Uniq-False* **by** *force*
next
case *True*
then obtain *vs* **where** *x-vs*: (*x*,*vs*) ∈ *set mpv* **by** *force*
with *r* **have** *uniq*: *UNIQ* (α ' *set vs*) **by** *auto*
from *split-list*[*OF x-vs*] **obtain** *bef aft* **where** *mpv*: *mpv* = *bef* @ (*x*,*vs*) #

aft **by** *auto*
from *dist*[*unfolded arg-cong*[*OF this*, *of map fst*]]
have *x*: *x* ∉ *fst* ‘ *set bef* ∪ *fst* ‘ *set aft* **by** *auto*
hence α ‘ *set vs* = *?vs* **unfolding** *match-of-var-form-list-def mpv* **by** *force*
with *uniq* **show** *?thesis* **by** *auto*
qed
qed
next
fix *mp*
assume *mp* ∈ *pat-list pp* **and** *uniq*: ∀ *x*. *UNIQ* {α *v* |*v*. (*Var v*, *Var x*) ∈ *mp*}
from *this*[*unfolded pp pat-list-def pat-of-var-form-list-def*]
obtain *mpv* **where** *mem*: *mpv* ∈ *set fvf* **and** *mp*: *mp* = *set* (*match-of-var-form-list mpv*) **by** *auto*
from *dist mem* **have** *dist*: *distinct* (*map fst mpv*) **by** *auto*
show ∃ *mpv*∈*set fvf*. ∀ (*x*, *vs*)∈*set mpv*. *UNIQ* (α ‘ *set vs*)
proof (*intro bexI*[*OF - mem*], *safe*)
fix *x vs*
assume (*x*,*vs*) ∈ *set mpv*
from *split-list*[*OF this*] **obtain** *bef aft* **where** *mpv*: *mpv* = *bef* @ (*x*,*vs*) # *aft* **by** *auto*
from *dist*[*unfolded arg-cong*[*OF this*, *of map fst*]]
have *x*: *x* ∉ *fst* ‘ *set bef* ∪ *fst* ‘ *set aft* **by** *auto*
from *uniq*[*rule-format*, *of x*]
have *UNIQ* {α *v* |*v*. (*Var v*, *Var x*) ∈ *mp*} **.**
also have {α *v* |*v*. (*Var v*, *Var x*) ∈ *mp*} = α ‘ *set vs*
unfolding *mp match-of-var-form-list-def mpv* **using** *x* **by** *force*
finally show *UNIQ* (α ‘ *set vs*) **.**
qed
qed
note *finite-var-form-pat-pat-complete*[*OF fvf*, *unfolded this*]
note *main* = *this*[*folded set-tvars-pat-list*]
show *?thesis* **unfolding** *main*
by (*intro all-cong*, *force split*: *prod.splits*)
qed

lemma *pat-complete-via-cnf*:
assumes *fvf*: *finite-var-form-pat C* (*pat-list pp*)
and *pp*: *pp* = *pat-of-var-form-list fvf*
and *dist*: *Ball* (*set fvf*) (*distinct* ∘ *map fst*)
and *cnf*: *cnf* = *map* (*map snd*) *fvf*
shows *pat-complete C* (*pat-list pp*) ⟷
(∀ α. (∀ *v* ∈ *set* (*concat* (*concat cnf*)). α *v* < *card-of-sort C* (*snd v*)) ⟶
(∃ *c* ∈ *set cnf*. ∀ *vs* ∈ *set c*. *UNIQ* (α ‘ *set vs*)))
unfolding *finite-var-form-pat-pat-complete-list*[*OF fvf pp dist*] *cnf*[*symmetric*]
proof (*intro all-cong1 arg-cong*[*of - - λ x*. *x* ⟶ -] *ball-cong refl*)
show *set* (*tvars-pat-list pp*) = *set* (*concat* (*concat cnf*)) **unfolding** *tvars-pat-list-def cnf pp*
by (*force simp*: *tvars-match-list-def pat-of-var-form-list-def match-of-var-form-list-def*)

qed

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

Various well-formed predicates for intermediate results

definition *wf-ts* :: ('*f*, *nat* × '*s*) *term list* ⇒ *bool* **where**
wf-ts ts = (*ts* ≠ [] ∧ *distinct ts* ∧ (∀ *j* < *length ts*. ∀ *i* < *j*. *conflicts* (*ts* ! *i*) (*ts* ! *j*) ≠ *None*))

definition *wf-ts2* :: ('*f*, *nat* × '*s*) *term list* ⇒ *bool* **where**
wf-ts2 ts = (*length ts* ≥ *2* ∧ *distinct ts* ∧ (∀ *j* < *length ts*. ∀ *i* < *j*. *conflicts* (*ts* ! *i*) (*ts* ! *j*) ≠ *None*))

definition *wf-ts3* :: ('*f*, *nat* × '*s*) *term list* ⇒ *bool* **where**
wf-ts3 ts = (∃ *t* ∈ *set ts*. *is-Var t*)

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

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

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

definition *wf-rx3* :: ('*f*,'*v*,'*s*)*match-problem-rx* ⇒ *bool* **where**
wf-rx3 rx = (*wf-rx2 rx* ∧ (*improved* ⟶ *snd rx* ∨ (*Ball* (*snd* ‘ *set* (*fst rx*)) *wf-ts3*)))

definition *wf-lr* :: ('*f*,'*v*,'*s*)*match-problem-lr* ⇒ *bool*
where *wf-lr pair* = (*case pair of* (*lx*,*rx*) ⇒ *wf-lx lx* ∧ *wf-rx rx*)

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

definition *wf-lr3* :: ('*f*,'*v*,'*s*)*match-problem-lr* ⇒ *bool*
where *wf-lr3 pair* = (*case pair of* (*lx*,*rx*) ⇒ *wf-lx lx* ∧ (*if lx* = [] *then wf-rx3 rx else wf-rx rx*))

definition *wf-pat-lr* :: ('*f*,'*v*,'*s*)*pat-problem-lr* ⇒ *bool* **where**
wf-pat-lr mps = (*Ball* (*set mps*) (λ *mp*. *wf-lr3 mp* ∧ ¬ *empty-lr mp*))

definition *wf-pat-lx* :: ('*f*,'*v*,'*s*)*pat-problem-lx* ⇒ *bool* **where**
wf-pat-lx mps = (*Ball* (*set mps*) (λ *mp*. *ll-mp* (*mp-list* (*mp-lx mp*)) ∧ *wf-lx mp* ∧ *mp* ≠ [])))

lemma *wf-rx-mset*: **assumes** *mset rx = mset rx′*
shows *wf-rx (rx,b) = wf-rx (rx′,b)*
proof −
from *assms* **have** *set*: *set rx = set rx′* **by** (*metis mset-eq-setD*)
show *?thesis*
unfolding *wf-rx-def fst-conv snd-conv mp-rx-def set*
apply (*intro conj-cong refl mset-eq-imp-distinct-iff*
arg-cong2[*of - - - - (=)*]
arg-cong[*of - - inf-var-conflict*])
subgoal using *assms* **by** *simp*
subgoal by (*auto simp*: *List.maps-def set*)
done
qed

lemma *wf-rx2-mset*: **assumes** *mset rx = mset rx′*
shows *wf-rx2 (rx,b) = wf-rx2 (rx′,b)*
proof −
from *assms* **have** *set*: *set rx = set rx′* **by** (*metis mset-eq-setD*)
show *?thesis*
unfolding *wf-rx2-def fst-conv snd-conv mp-rx-def set*
apply (*intro conj-cong refl mset-eq-imp-distinct-iff*
arg-cong2[*of - - - - (=)*]
arg-cong[*of - - inf-var-conflict*])
subgoal using *assms* **by** *simp*
subgoal by (*auto simp*: *List.maps-def set*)
done
qed

lemma *wf-lr2-mset*: **assumes** *mset rx = mset rx′*
shows *wf-lr2 (lx,(rx,b)) = wf-lr2 (lx,(rx′,b))*
using *assms*
unfolding *wf-lr2-def split wf-rx2-mset*[*OF assms*] *wf-rx-mset*[*OF assms*]
by *simp*

lemma *mp-lr-mset*: **assumes** *mset rx = mset rx′*
shows *mp-lr (lx,(rx,b)) = mp-lr (lx,(rx′,b))*
unfolding *mp-lr-def split mp-rx-def List.maps-def mset-concat-union* **using** *assms*

by *auto*

lemma *mp-list-lr*: *mp-list* (*mp-lr-list mp*) = *mp-lr mp*
unfolding *mp-lr-list-def mp-lr-def*
by (*cases mp*, *auto simp*: *mp-rx-def mp-rx-list-def*)

lemma *pat-mset-list-lr*: *pat-mset-list* (*map mp-lr-list pp*) = *pat-lr pp*
unfolding *pat-lr-def pat-mset-list-def map-map o-def mp-list-lr* **by** *simp*

lemma *size-term-0*[*simp*]: *size* (*t* :: (*'f*,*'v*)*term*) > *0*
by (*cases t*, *auto*)

lemma *wf-ts-no-conflict-alt-def*: (∀ *j* < *length ts*. ∀ *i* < *j*. *conflicts* (*ts* ! *i*) (*ts* ! *j*)
≠ *None*)
⟷ (∀ *s t*. *s* ∈ *set ts* ⟶ *t* ∈ *set ts* ⟶ *conflicts s t* ≠ *None*) (**is** *?l* = *?r*)
proof
assume *?l*
note *l* = *this*[*rule-format*]
show *?r*
proof (*intro allI impI*)
fix *s t*
assume *s* ∈ *set ts t* ∈ *set ts*
then obtain *i j* **where** *ij*: *i* < *length ts j* < *length ts*
and *st*: *s* = *ts* ! *i t* = *ts* ! *j* **unfolding** *set-conv-nth* **by** *auto*
then consider (*lt*) *i* < *j* | (*eq*) *i* = *j* | (*gt*) *j* < *i* **by** *linarith*
thus *conflicts s t* ≠ *None*
proof *cases*
case *lt*
show *?thesis* **using** *l*[*OF ij*(*2*) *lt*] **unfolding** *st* **by** *auto*
next
case *eq*
show *?thesis* **unfolding** *st eq* **by** *simp*
next
case *gt*
show *?thesis* **using** *l*[*OF ij*(*1*) *gt*] *conflicts-sym*[*of s t*] **unfolding** *st*
by (*simp add*: *option.rel-sel*)
qed
qed
next
assume *?r*
note *r* = *this*[*rule-format*]
show *?l*
proof (*intro allI impI*)
fix *j i*
assume *j* < *length ts i* < *j*
hence *ts* ! *i* ∈ *set ts ts* ! *j* ∈ *set ts* **by** (*auto simp*: *set-conv-nth*)
from *r*[*OF this*] **show** *conflicts* (*ts* ! *i*) (*ts* ! *j*) ≠ *None*
by *auto*
qed
qed

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'* ⟹ (→$_m$)** (*add-mset* (*t*,*Var x*) (*mp-lr mp* + *M*)) (*mp-lr*

```
(ls,rx') + M) ∧ wf-rx rx'
  ∧ lvars-mp (add-mset (t,Var x) (mp-lr mp + M)) ⊇ lvars-mp (mp-lr (ls,rx') +
M)
    res = None ⟹ 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' ∈ 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) ∈# mp-rx rxb
       unfolding mp-rx-def rxb by auto
    hence (t',Var x) ∈# 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) ∧ wf-rx rx'
     ∧ lvars-mp (mp-lr (ls, rx') + M) ⊆ lvars-mp (add-mset (t, Var x) (mp-lr mp
+ M))
    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) ∉ 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*) ∈ *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 simp*: *lvars-mp-def*)
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 ⟹
    (res = Some mp' ⟶ (→m)** (mp-list mp + M) (mp-lr mp' + M) ∧ wf-lr mp'

      ∧ lvars-mp (mp-list mp + M) ⊇ lvars-mp (mp-lr mp' + M))
  ∧ (res = None ⟶ (∃ mp'. (→m)** (mp-list mp + M) mp' ∧ 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]
    have lvars: lvars-mp (mp-list ((Fun f ts, Fun g ls) # mp) + M) ⊇ lvars-mp
(mp-list (zip ts ls @ mp) + M)
      by (auto simp: lvars-mp-def dest: set-zip-rightD)
    from IH step subset-trans[OF - lvars]
    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: (→m)** (mp-list ((t, Var y) # mp) + M) (mp-lr ( lx, rx) + add-mset
(t, Var y) M)
      wf-lr ( lx, rx)
      lvars-mp (mp-lr (lx, rx) + add-mset (t, Var y) M)
       ⊆ lvars-mp (mp-list mp + add-mset (t, Var y) M) 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: (→m)** (mp-lr ( lx, rx) + add-mset (t, Var y) M) (mp-lr ( lx,
rx') + M)
        and lvars: lvars-mp (mp-lr (lx, rx') + M) ⊆ lvars-mp (add-mset (t, Var y)
(mp-lr (lx, rx) + M))
        by auto
      from IH(1) steps
     have steps: (→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 lvars IH(3) show ?thesis by auto
    qed
  qed
```

qed

lemma *match-decomp-lin-impl*: *match-decomp-lin-impl mp = res* $\Longrightarrow$ *ll-mp* (*mp-list mp* + *M*) $\Longrightarrow$
  (*res = Some mp′* $\longrightarrow$ $(\rightarrow_m)^{**}$ (*mp-list mp* + *M*) (*mp-list* (*mp-lx mp′*) + *M*) $\wedge$ *wf-lx mp′* $\wedge$ *ll-mp* (*mp-list* (*mp-lx mp′*) + *M*))
 $\wedge$ (*res = None* $\longrightarrow$ ($\exists$ *mp′*. $(\rightarrow_m)^{**}$ (*mp-list mp* + *M*) *mp′* $\wedge$ *match-fail mp′*))
proof (*induct mp arbitrary*: *res M mp′ rule*: *match-decomp-lin-impl.induct*)
 case *1*
 thus *?case* **by** (*auto simp*: *mp-lr-def wf-lx-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*: *match-decomp-lin-impl* (*zip ts ls* @ *mp*) = *res* **by** *auto*
  have *imag-snd*: *image-mset snd* (*mp-list* (*zip ts ls*)) = *mset ls* **using** *True*
   by *simp* (*metis map-snd-zip mset-map*)
  have *vars-mp-mset* (*mp-list* ((*Fun f ts*, *Fun g ls*) # *mp*) + *M*)
   = *vars-term-ms* (*Fun g ls*) + *vars-mp-mset* (*mp-list mp* + *M*)
   unfolding *vars-mp-mset-def* **by** *auto*
  also have *vars-term-ms* (*Fun g ls*) = *vars-mp-mset* (*mp-list* (*zip ts ls*))
   unfolding *vars-mp-mset-def image-mset.comp*[*symmetric*]
   unfolding *o-def imag-snd* **by** *simp*
  finally have *vars-mp-mset* (*mp-list* ((*Fun f ts*, *Fun g ls*) # *mp*) + *M*)
     = *vars-mp-mset* (*mp-list* (*zip ts ls* @ *mp*) + *M*)
   unfolding *vars-mp-mset-def* **by** *auto*
  with *2*(*3*) **have** *ll*: *ll-mp* (*mp-list* (*zip ts ls* @ *mp*) + *M*) **unfolding** *ll-mp-def* **by** *auto*
  note *IH* = *2*(*1*)[*OF True res ll*, *of mp′*]
  note *step* = *match-decompose*[*OF True*, *of mp-list mp* + *M*, *folded id id2*]
  from *IH step subset-trans*
  show *?thesis* **by** (*meson converse-rtranclp-into-rtranclp*)
 qed
next
 case (*3 x g ls mp res M mp′*)
 note *res* = *3*(*2*)[*unfolded match-decomp-lin-impl.simps*]

from *3(3)* **have** *ll*: *ll-mp* (*mp-list mp* + *add-mset* (*Var x*, *Fun g ls*) *M*) **by** *simp*
note *IH* = *3(1)*[*OF* - *ll*]
show *?case*
proof (*cases match-decomp-lin-impl mp*)
case *None*
from *IH*[*OF None*] *None res* **show** *?thesis* **by** *auto*
next
case (*Some mpx*)
then obtain *lx* **where** *decomp*: *match-decomp-lin-impl mp* = *Some lx* **by** (*cases mpx*, *auto*)
from *res*[*unfolded decomp option.simps split*] **have** *res*: *res* = *Some* ((*x*, *Fun g ls*) # *lx*) **by** *auto*
from *IH*[*OF decomp*, *of lx*] *res*
show *?thesis* **by** (*auto simp*: *wf-lx-def*)
qed
next
case (*4 t y mp res M mp'*)
note *res* = *4(2)*[*unfolded match-decomp-lin-impl.simps*]
have *vars-mp-mset* (*mp-list mp* + *M*) ⊆# *vars-mp-mset* (*mp-list* ((*t*, *Var y*) # *mp*) + *M*)
unfolding *vars-mp-mset-def* **by** *auto*
with *4(3)* **have** *ll-new*: *ll-mp* (*mp-list mp* + *M*) **unfolding** *ll-mp-def*
by (*meson dual-order.trans subseteq-mset-def*)
have *mp-list* ((*t*, *Var y*) # *mp*) + *M* = *add-mset* (*t*, *Var y*) (*mp-list mp* + *M*) **by** *auto*
also have ... →$_m$ *mp-list mp* + *M*
proof (*rule match-match*)
from *4(3)*[*unfolded ll-mp-def*]
have *count* (*vars-mp-mset* (*mp-list* ((*t*, *Var y*) # *mp*) + *M*)) *y* ≤ *1* **by** *auto*
hence *count* (*vars-mp-mset* (*mp-list mp* + *M*)) *y* = *0*
unfolding *vars-mp-mset-def* **by** *auto*
hence *y* ∉# *vars-mp-mset* (*mp-list mp* + *M*)
by (*simp add*: *not-in-iff*)
hence *y* ∉ *set-mset* (*vars-mp-mset* (*mp-list mp* + *M*)) **by** *blast*
also have *set-mset* (*vars-mp-mset* (*mp-list mp* + *M*)) = ⋃ (*vars* ‘ *snd* ‘ *mp-mset* (*mp-list mp* + *M*))
unfolding *vars-mp-mset-def o-def* **by** *auto*
finally show *y* ∉ ⋃ (*vars* ‘ *snd* ‘ *mp-mset* (*mp-list mp* + *M*)) **by** *auto*
qed
finally have *mp-list* ((*t*, *Var y*) # *mp*) + *M* →$_m$ *mp-list mp* + *M* **.**
note *step* = *converse-rtranclp-into-rtranclp*[*of mp-step-mset*, *OF this*]
note *IH* = *4(1)*[*OF* - *ll-new*]
show *?case*
proof (*cases match-decomp-lin-impl mp*)
case *None*
with *IH*[*OF None*] *res step* **show** *?thesis* **by** *fastforce*
next
case (*Some mpx*)
with *IH*[*OF Some*, *of mpx*] *res step* **show** *?thesis* **by** *fastforce*

qed
qed

lemma *pat-inner-lin-impl*: **assumes** *pat-inner-lin-impl p pd = res*
and *wf-pat-lx pd* ∀ *mp* ∈ *set p. ll-mp* (*mp-list mp*)
and *tvars-pat* (*pat-mset* (*pat-mset-list p* + *pat-lx pd*)) ⊆ *V*
shows *res = None* ⟹ (*add-mset* (*pat-mset-list p* + *pat-lx pd*) *P*, *P*) ∈ ⇛$^+$
and *res = Some p′* ⟹ (*add-mset* (*pat-mset-list p* + *pat-lx pd*) *P*, *add-mset* (*pat-lx p′*) *P*) ∈ ⇛*
∧ *wf-pat-lx p′* ∧ *tvars-pat* (*pat-mset* (*pat-lx p′*)) ⊆ *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-lx pd*
have *id*: *pat-mset-list* (*mp* # *p*) + *pat-lx pd* = *add-mset* (*mp-list mp*) *?p* **unfolding** *pat-mset-list-def* **by** *auto*
from *Cons*(*4*) **have** *ll-mp* (*mp-list mp* + {#}) **by** *auto*
note *match* = *match-decomp-lin-impl*[*OF - this*]
note *res* = *Cons*(*2*)[*unfolded pat-inner-lin-impl.simps*]
from *Cons*(*4*) **have** *llp*: ∀ *mp*∈*set p. ll-mp* (*mp-list mp*)
and *ll-mp*: *ll-mp* (*mp-list mp*) **by** *auto*
show *?case*
proof (*cases match-decomp-lin-impl mp*)
case (*Some mp′*)
from *match*[*OF this*, *of mp′*]
have *steps*: (→$_m$)** (*mp-list mp*) (*mp-list* (*mp-lx mp′*)) **and** *wf*: *wf-lx mp′*
and *ll-mp′*: *ll-mp* (*mp-list* (*mp-lx mp′*)) **by** *auto*
from *mp-step-mset-steps-vars*[*OF steps*]
have *tvars*: *tvars-match* (*mp-mset* (*mp-list* (*mp-lx mp′*))) ⊆ *tvars-match* (*mp-mset* (*mp-list mp*)) **by** *auto*
note *Psteps* = *mp-step-mset-cong*[*OF steps*, *of ?p P*, *folded id*]
note *res* = *res*[*unfolded Some option.simps*]
show *?thesis*
proof (*cases mp′* = [])
case *True*
with *res* **have** *res*: *res = None* **by** *auto*
from *True* **have** *empty*: *mp-list* (*mp-lx mp′*) = {#} **by** *auto*
have (*add-mset* (*add-mset* (*mp-list* (*mp-lx mp′*)) *?p*) *P*, {#} + *P*) ∈ ⇛
unfolding *empty* **unfolding** *P-step-def*
by (*standard*, *unfold split*, *rule P-simp-pp*, *rule pat-remove-pp*)
with *Psteps*
show *?thesis* **using** *res* **by** *auto*
next
case *False*
with *res* **have** *res*: *pat-inner-lin-impl p* (*mp′* # *pd*) = *res* **by** *auto*
have *wf-pat-lx* (*mp′* # *pd*) **using** *wf ll-mp′ Cons*(*3*) *False*

```
      unfolding wf-pat-lx-def by auto
    note IH = Cons(1)[OF res this llp, of p']
    have tvars: tvars-pat (pat-mset (pat-mset-list p + pat-lx (mp' # pd))) ⊆ V
      using tvars Cons(5) unfolding tvars-pat-def
      by (auto simp: pat-lx-def pat-mset-list-def)
    note IH = IH[OF this]
    define I1 where I1 = add-mset (pat-mset-list p + pat-lx (mp' # pd)) P
   define I2 where I2 = add-mset (add-mset (mp-list (mp-lx mp')) (pat-mset-list
p + pat-lx pd)) P
    have I2 = I1 unfolding I1-def I2-def by (auto simp: pat-lx-def)
    define S where S = add-mset (pat-mset-list (mp # p) + pat-lx pd) P
    define E where E = add-mset (pat-lx p') P
    from IH Psteps show ?thesis
    unfolding I1-def[symmetric] I2-def[symmetric] S-def[symmetric] E-def[symmetric]
      unfolding ‹I2 = I1› by auto
  qed
 next
  case None
  from match[OF None] obtain mp' where
    msteps: (→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]
  note tvars = mp-step-mset-steps-vars[OF msteps]
  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-lx pd) P,
add-mset ?p P) ∈ P-step^* by auto
  from res[unfolded None option.simps]
  have res: pat-inner-lin-impl p pd = res by auto
  note IH = Cons(1)[OF res Cons(3) llp, of p']
  have tvars-pat (pat-mset (pat-mset-list p + pat-lx pd)) ⊆ V
    using Cons(5) unfolding tvars-pat-def
    by (auto simp: pat-lx-def pat-mset-list-def)
  from IH[OF this] steps tvars
  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)

lemma vars-mp-mset-subst: vars-mp-mset (mp-list (subst-match-problem-list τ mp))

 = vars-mp-mset (mp-list mp)
 unfolding vars-mp-mset-def subst-match-problem-list-def subst-left-def
 by (simp add: image-mset.comp[symmetric], intro
  arg-cong[of - - λ xs. ∑# (image-mset vars-term-ms xs)])
  (induct mp, auto)
```

lemma *subst-conversion*: *map* (λτ. *subst-pat-problem-mset* τ (*pat-mset-list p*)) *xs*
=
map pat-mset-list (*map* (λτ. *subst-pat-problem-list* τ *p*) *xs*)
unfolding *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*)

lemma *ll-mp-subst*: *ll-mp* (*mp-list* (*subst-match-problem-list* τ *mp*)) = *ll-mp* (*mp-list mp*)
unfolding *ll-mp-def vars-mp-mset-subst* **by** *simp*

lemma *ll-pp-subst*: *ll-pp* (*subst-pat-problem-list* τ *p*) = *ll-pp p*
unfolding *ll-pp-def subst-pat-problem-list-def* **using** *ll-mp-subst*[*of* τ]
by *auto*

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

lemma *pat-lin-impl*:
assumes *pat-lin-impl n p* = *res*
and *vars*: *tvars-pat* (*pat-list p*) ⊆ {..<*n*} × *S*
and *linear*: *ll-pp p*
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*) ∈ ⇛⁺
∧ *tvars-pat* (⋃ (*pat-list* ‘ *set ps*)) ⊆ {..< *n* + *m*} × *S*
∧ *Ball* (*set ps*) *ll-pp*
proof (*atomize*(*full*), *goal-cases*)
case *1*
have *wf*: *wf-pat-lx* [] **unfolding** *wf-pat-lx-def* **by** *auto*
have *vars*: *tvars-pat* (*pat-mset* (*pat-mset-list p*)) ⊆ {..<*n*} × *S*
using *vars* **unfolding** *pat-mset-list* **by** *auto*
have *pat-mset-list p* + *pat-lx* [] = *pat-mset-list p* **unfolding** *pat-lx-def* **by** *auto*
note *pat-inner* = *pat-inner-lin-impl*[*OF refl wf*, *of p*, *unfolded this*, *OF linear*[*unfolded ll-pp-def*] *vars*]
note *res* = *assms*(*1*)[*unfolded pat-lin-impl-def*]
show *?case*
proof (*cases pat-inner-lin-impl p* [])
case *None*
from *pat-inner*(*1*)[*OF this*] *res*[*unfolded None option.simps*] *vars*
show *?thesis* **by** (*auto simp*: *tvars-pat-def*)
next
case (*Some p'*)
from *pat-inner*(*2*)[*OF Some*]
have *steps*: (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*pat-lx p'*) *P*) ∈ ⇛*
and *wf*: *wf-pat-lx p'*
and *varsp'*: *tvars-pat* (*pat-mset* (*pat-lx p'*)) ⊆ {..<*n*} × *S*
by *auto*

note *res* = *res*[*unfolded Some option.simps*]
show *?thesis*
proof (*cases p′*)
case *Nil*
with *res* **have** *res*: *res* = *None* **by** *auto*
from *Nil* **have** *pat-lx p′* = {#} **by** (*auto simp*: *pat-lx-def*)
hence *fail*: *pat-fail* (*pat-lx p′*)
using *pat-empty* **by** *auto*
from *fail res steps* **show** *?thesis* **by** *auto*
next
case (*Cons mp mps*)
from *wf*[*unfolded Cons wf-pat-lx-def*] **have** *mp*: *wf-lx mp mp* ≠ [] **by** *auto*
then obtain *f ts x mp′* **where** *mp* = (*x*,*Fun f ts*) # *mp′*
by (*cases mp*; *cases snd* (*hd mp*), *auto simp*: *wf-lx-def*)
note *Cons* = *Cons*[*unfolded this*]
from *Cons* **have** *id*: (*p′* = []) = *False* **by** *auto*
define *p′l* **where** *p′l* = *map mp-lx p′*
note *res* = *res*[*unfolded Cons list.sel fst-conv*, *folded Cons*, *unfolded id if-False Let-def*]
from *res* **have** *res*: *res* = *Some* (*map* (λτ. *subst-pat-problem-list* τ *p′l*) (τ*s-list n x*))
by (*auto simp*: *p′l-def*)
show *?thesis*
proof (*intro conjI impI*)
assume *res* = *Some ps*
with *res* **have** *ps-def*: *ps* = *map* (λτ. *subst-pat-problem-list* τ *p′l*) (τ*s-list n x*) **by** *auto*
have *id*: *pat-lx p′* = *pat-mset-list p′l* **unfolding** *p′l-def pat-lx-def pat-mset-list-def*
by *auto*
have *ll*: *ll-pp p′l* **unfolding** *p′l-def* **using** *wf* **unfolding** *wf-pat-lx-def ll-pp-def* **by** *auto*
thus *Ball* (*set ps*) *ll-pp* **unfolding** *ps-def* **using** *ll-pp-subst* **by** *auto*
have *subst*: *map* (λτ. *subst-pat-problem-mset* τ (*pat-lx p′*)) (τ*s-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*)
have *step*: (*add-mset* (*pat-lx p′*) *P*, *mset* (*map pat-mset-list ps*) + *P*) ∈ ⇛
unfolding *P-step-def*
proof (*standard*, *unfold split*, *intro P-simp-pp*)
note *x* = *Some*[*unfolded find-var-def*]
have *disj*: *tvars-disj-pp* {*n*..<*n* + *m*} (*pat-mset* (*pat-lx p′*))
using *varsp′* **unfolding** *tvars-pat-def tvars-disj-pp-def tvars-match-def* **by** *force*
obtain *mp″ p″* **where** *expand*: *pat-lx p′* = *add-mset* (*add-mset* (*Var x*, *Fun f ts*) *mp″*) *p″*
unfolding *Cons pat-lx-def* **by** *auto*

```
        have pat-lx p' ⇒m mset (map (λτ. subst-pat-problem-mset τ (pat-lx p'))
(τs-list n x)) (is - ⇒m ?ps)
          using pat-instantiate[OF disj[unfolded expand], folded expand, of x Fun f
ts]
          by auto
        also have ?ps = mset (map pat-mset-list ps)
          unfolding ps-def id unfolding subst-conversion ..
        finally show pat-lx p' ⇒m mset (map pat-mset-list ps) by auto
      qed
      with steps
       show (add-mset (pat-mset-list p) P, mset (map pat-mset-list ps) + P) ∈
⇛+
        by auto
      show tvars-pat (⋃ (pat-list ‘ set ps)) ⊆ {..<n + m} × S
      proof (safe del: conjI)
        fix yn ι
        assume (yn,ι) ∈ tvars-pat (⋃ (pat-list ‘ set ps))
        then obtain pi mp
          where pi: pi ∈ set ps
            and mp: mp ∈ set pi and y: (yn,ι) ∈ tvars-match (set mp)
          unfolding tvars-pat-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: τ ∈ set (τs-list n x) and pi: pi = map (map (subst-left
τ)) p'l by auto
        from tau[unfolded τs-list-def]
        obtain info where infoCl: info ∈ set (Cl (snd x)) and tau: τ = τc n x
info by auto
        from Cl-len[of snd x] this(1) have len: length (snd info) ≤ m by force
        from mp[unfolded pi set-map] obtain mp' where mp': mp' ∈ set p'l and
mp: mp = map (subst-left τ) mp' by auto
        from y[unfolded mp tvars-match-def image-comp o-def set-map]
       obtain pair where ∗: pair ∈ set mp' (yn,ι) ∈ 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) ∈ set mp' and y: (yn,ι) ∈ vars (s · τ)
unfolding subst-left-def by auto
        from y[unfolded vars-term-subst, simplified]
        obtain z where z: z ∈ vars s and y: (yn,ι) ∈ vars (τ z) by auto
        obtain f ss where info: info = (f,ss) by (cases info, auto)
        with len have len: length ss ≤ m by auto
        define ts :: ('f,-)term list where ts = map Var (zip [n..<n + length ss]
ss)
        from tau[unfolded τc-def info split]
        have tau: τ = subst x (Fun f ts) unfolding ts-def by auto
        from infoCl[unfolded Cl info]
        have f: f : ss → snd x in C by auto
        from C-sub-S[OF this] have ssS: set ss ⊆ S by simp
        from ssS
```

```
        have vars (Fun f ts) ⊆ {..< n + length ss} × S unfolding ts-def by (auto
simp: set-zip)
         also have ... ⊆ {..< n + m} × S using len by auto
         finally have subst: vars (Fun f ts) ⊆ {..< n + m} × S by auto
         show yn ∈ {..<n + m} ∧ ι ∈ S
         proof (cases z = x)
           case True
           with y subst tau show ?thesis by force
         next
           case False
           hence τ z = Var z unfolding tau by (auto simp: subst-def)
           with y have z = (yn,ι) by auto
           with z have y: (yn,ι) ∈ vars s by auto
          with st have (yn,ι) ∈ tvars-match (set mp') unfolding tvars-match-def
by force
           with mp' have (yn,ι) ∈ tvars-pat (set ‘ set p'l) unfolding tvars-pat-def
by auto
           also have ... = tvars-pat (pat-mset (pat-mset-list p'l))
          by (rule arg-cong[of - - tvars-pat], auto simp: pat-mset-list-def image-comp)
          also have ... = tvars-pat (pat-mset (pat-lx p')) unfolding id[symmetric]
by simp
           also have ... ⊆ {..<n} × S using varsp' .
           finally show ?thesis by auto
         qed
        qed
      qed (insert res, auto)
    qed
  qed
qed

lemma pats-mset-list: pats-mset (pats-mset-list ps) = pat-list ‘ set ps
  unfolding pat-list-def pat-mset-list-def o-def set-mset-mset set-map
      mset-map image-comp set-image-mset by simp


lemma pats-lin-impl: assumes ∀ p ∈ set ps. tvars-pat (pat-list p) ⊆ {..<n} × S
  and Ball (set ps) ll-pp
  and ∀ pp ∈ pat-list ‘ set ps. wf-pat pp
  shows pats-lin-impl n ps = pats-complete C (pat-list ‘ set ps)
proof (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)
  note IH = 1(1)
  note ll = 1(3)
  note wf = 1(4)
  note simps = pats-lin-impl.simps[of n ps]
  show ?case
  proof (cases ps)
    case Nil
```

show *?thesis* **unfolding** *simps* **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* = *simps*[*unfolded Cons list.simps*, *folded Cons*]
from *1*(*2*)[*rule-format*, *of p*] *Cons* **have** *tvars-pat* (*pat-list p*) ⊆ {..<*n*} × *S* **by** *auto*
note *pat-impl* = *pat-lin-impl*[*OF refl this*]
from *ll Cons* **have** *ll-pp p* **by** *auto*
note *pat-impl* = *pat-impl*[*OF this*, **where** *P* = (*pats-mset-list ps1*), *folded id*]
let *?step* = (⇛) :: (('*f*,'*v*,'*s*)*pats-problem-mset* × ('*f*,'*v*,'*s*)*pats-problem-mset*)*set*

from *wf* **have** *wf-pats* (*pat-list* ' *set ps*) **unfolding** *wf-pats-def* **by** *auto*
note *steps-to-equiv* = *P-steps-pcorrect*[*OF this*[*folded pats-mset-list*]]
show *?thesis*
proof (*cases pat-lin-impl n p*)
case *None*
with *res* **have** *res*: *pats-lin-impl n ps* = *False* **by** *auto*
from *pat-impl*(*1*)[*OF None*]
obtain *p'* **where** *steps*: (*pats-mset-list ps*, *add-mset p'* (*pats-mset-list ps1*)) ∈ ⇛* **and** *fail*: *pat-fail p'*
by *auto*
show *?thesis*
proof (*cases add-mset p'* (*pats-mset-list ps1*) = *bottom-mset*)
case *True*
with *res P-steps-pcorrect*[*OF - steps*, *unfolded pats-mset-list*] *wf*
show *?thesis* **by** (*auto simp*: *wf-pats-def*)
next
case *False*
from *P-failure*[*OF fail False*]
have (*add-mset p'* (*pats-mset-list ps1*), *bottom-mset*) ∈ ⇛ **unfolding** *P-step-def* **by** *auto*
with *steps* **have** (*pats-mset-list ps*, *bottom-mset*) ∈ ⇛* **by** *auto*
from *steps-to-equiv*[*OF this*] *res* **show** *?thesis* **unfolding** *pats-mset-list* **by** *simp*
qed
next
case (*Some ps2*)
with *res* **have** *res*: *pats-lin-impl n ps* = *pats-lin-impl* (*n* + *m*) (*ps2* @ *ps1*) **by** *auto*
from *pat-impl*(*2*)[*OF Some*]
have *steps*: (*pats-mset-list ps*, *mset* (*map pat-mset-list* (*ps2* @ *ps1*))) ∈ ⇛+
and *vars*: *tvars-pat* (⋃ (*pat-list* ' *set ps2*)) ⊆ {..<*n* + *m*} × *S*
and *ll*: *Ball* (*set ps2*) *ll-pp*
by *auto*
have *vars*: ∀ *p*∈*set* (*ps2* @ *ps1*). *tvars-pat* (*pat-list p*) ⊆ {..<*n* + *m*} × *S*
proof
fix *p*

```
      assume p ∈ set (ps2 @ ps1)
      hence p ∈ set ps2 ∨ p ∈ set ps1 by auto
      thus tvars-pat (pat-list p) ⊆ {..<n+ m} × S
      proof
        assume p ∈ set ps2
        hence tvars-pat (pat-list p) ⊆ tvars-pat (⋃ (pat-list ‘ set ps2))
          unfolding tvars-pat-def by auto
        with vars show ?thesis by auto
      next
        assume p ∈ set ps1
        hence p ∈ set ps unfolding Cons by auto
        from 1(2)[rule-format, OF this] show ?thesis by auto
      qed
    qed
    note steps-equiv = steps-to-equiv[OF trancl-into-rtrancl[OF steps]]
     from steps-equiv have wf-pats (pats-mset (mset (map pat-mset-list (ps2 @
ps1)))) by auto
     hence wf2: Ball (pat-list ‘ set (ps2 @ ps1)) wf-pat unfolding wf-pats-def
pats-mset-list[symmetric]
      by auto
    have pats-lin-impl n ps = pats-lin-impl (n + m) (ps2 @ ps1) unfolding res
by simp
    also have ... = pats-complete C (pat-list ‘ set (ps2 @ ps1))
    proof (rule IH[OF - vars - wf2])
      show (ps, ps2 @ ps1) ∈ inv-image (⇛+) pats-mset-list
        using steps by auto
      show ∀ p∈set (ps2 @ ps1). ll-pp p using ll 1(3) Cons by auto
    qed
    also have ... = pats-complete C (pat-list ‘ set ps) using steps-equiv
      unfolding pats-mset-list[symmetric] by auto
    finally show ?thesis .
  qed
 qed
qed

corollary pat-complete-lin-impl:
 assumes wf: snd ‘ ⋃ (vars ‘ fst ‘ set (concat (concat P))) ⊆ S
 and left-linear: Ball (set P) ll-pp
 shows pat-complete-lin-impl (P :: ('f,'v,'s)pats-problem-list) ⟷ pats-complete
C (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-match-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: ∀ p∈set P. tvars-pat (pat-list p) ⊆ {..<n} × S
 proof (safe)
  fix p x ι
```

assume *p*: *p* ∈ *set P* **and** *xp*: (*x*,*ι*) ∈ *tvars-pat* (*pat-list p*)
hence *x* ∈ *set ?l* **unfolding** *List.maps-def tvars-pat-def tvars-match-def pat-list-def*

by *force*
from *max-list*[*OF this*] **have** *x* < *n* **unfolding** *n-def* **by** *auto*
thus *x* < *n* **by** *auto*
from *xp p wf*
show *ι* ∈ *S* **by** (*auto simp*: *wf-pat-iff*)
qed
have *pat-complete-lin-impl P* = *pats-lin-impl n P*
unfolding *pat-complete-lin-impl-def Let-def n-def* **by** *auto*
from *pats-lin-impl*[*OF n left-linear wf*, *folded this*]
show *?thesis* **by** *auto*
qed

lemma *match-var-impl*: **assumes** *wf*: *wf-lr mp*
and *match-var-impl mp* = (*xs*,*mpFin*)
shows (→$_m$)** (*mp-lr mp*) (*mp-lr mpFin*)
and *wf-lr2 mpFin*
and *lvars-mp* (*mp-lr mp*) ⊇ *lvars-mp* (*mp-lr mpFin*)
and *set xs* = *lvars-mp* (*mp-list* (*mp-lx* (*fst mpFin*)))
proof −
note [*simp*] = *List.maps-def*
let *?mp′* = *snd* (*match-var-impl mp*)
have *mpFin*: *mpFin* = *?mp′* **using** *assms*(*2*) **by** *auto*
from *assms* **obtain** *xl rx b* **where** *mp3*: *mp* = (*xl*,(*rx*,*b*)) **by** (*cases mp*, *auto*)
from *assms*(*2*) **have** *xs-def*: *xs* = *remdups* (*List.maps* (*vars-term-list o snd*) *xl*)

unfolding *match-var-impl-def mp3 split Let-def* **by** *auto*
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* ⟹ *?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 .
  have wf-lr2 ?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
  thus wf-lr2 mpFin unfolding mpFin .
  {
   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*: $x \notin fst$ ‘ *set rx1* $\cup$ *fst* ‘ *set rx2* **by** *auto*
define mp'' **where** $mp'' =$ *mp-rx* (*filter* (*Not* $\circ$ *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 $\exists$ *x mp''*. *xt* = *Var x* $\wedge$ *deleted* = *add-mset* (*t*, *Var x*) $mp'' \wedge x \notin \bigcup$ (*vars* ‘ *snd* ‘ (*mp-mset* mp'' $\cup$ *mp-mset* mp'))
proof (*intro exI conjI*, *rule xt*, *rule eq*, *intro notI*)
assume $x \in \bigcup$ (*vars* ‘ *snd* ‘ (*mp-mset* mp'' $\cup$ *mp-mset* mp'))
then obtain *s t'* **where** *st*: (*s*,*t'*) $\in$ *mp-mset* (mp' + mp'') **and** *xt*: $x \in$ *vars t'* **by** *force*
from *xrx* **have** (*s*,*t'*) $\notin$ *mp-mset* mp'' **using** *xt* **unfolding** *mp''-def mp-rx-def* **by** *force*
with *st* **have** (*s*,*t'*) $\in$ *mp-mset* mp' **by** *auto*
with *xxs* **have** (*s*, *t'*) $\in\#$ *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* $\neq$ [] $\vee$ *x* $\in$ *set xs*)› *fst-conv image-eqI prod.inject rx set-ConsD set-append ts xrx*)
qed
} **note** *lin-vars* = *this*
show ($\rightarrow_m$)** (*mp-lr mp*) (*mp-lr mpFin*) **unfolding** *mpFin 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*) $\in\#$ *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*: $x \notin \bigcup$ (*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* + $mp' \rightarrow_m$ (*deleted* + mp') .
have *two*: ($\rightarrow_m$)** (*deleted* + mp') mp'
proof (*rule add*(*1*), *goal-cases*)
case (*1 s yt*)
hence (*s*,*yt*) $\in\#$ *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 $\notin \bigcup$ (*vars* ‘ *snd* ‘ (*mp-mset* mp'' $\cup$ *mp-mset* mp'))
by *auto*
from *1*[*unfolded yt*] **have** *y* $\in \bigcup$ (*vars* ‘ *snd* ‘ (*mp-mset* (*deleted* + mp'))) **by** *force*
with *x* **have** *x* $\neq$ *y* **by** *auto*
with *pair yt* **have** *pair* $\neq$ (*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
 show lvars-mp (mp-lr mpFin) ⊆ lvars-mp (mp-lr mp)
  unfolding mp mp' deleted-def mp'-def mpFin
  by (auto simp: lvars-mp-def mp-lr-def)
 show set xs = lvars-mp (mp-list (mp-lx (fst mpFin)))
  unfolding mpFin
  unfolding xs-def lvars-mp-def mp3
  unfolding match-var-impl-def split snd-conv fst-conv Let-def
  by auto
qed

lemma match-steps-impl: assumes match-steps-impl mp = res
 shows res = Some (xs,mp') ⟹ (→m)** (mp-list mp) (mp-lr mp') ∧ wf-lr2 mp'
   ∧ lvars-mp (mp-list mp) ⊇ lvars-mp (mp-lr mp')
   ∧ set xs = lvars-mp (mp-list (mp-lx (fst 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''
   and lsub: lvars-mp (mp-lr mp'') ⊆ lvars-mp (mp-list mp) by auto
  from res[unfolded Some] have res = Some (match-var-impl mp'') by auto
  with match-var-impl[OF wf] steps res lsub show ?thesis
   by (cases match-var-impl mp'', auto)
 qed
qed

lemma finite-sort-imp-finite-sort-vars:
 assumes t : σ in T(C,V)
 and x ∈ vars t
 and ¬ inf-sort σ
shows ¬ inf-sort (snd x)
 using assms
```

```
proof (induct)
  case (Fun f ts σs σ)
  from Fun obtain t where t ∈ set ts and x ∈ vars t by auto
  then obtain i where i: i < length ts and x: x ∈ vars (ts ! i) by (auto simp:
set-conv-nth)
  from Fun(2)[unfolded list-all2-conv-all-nth]
  have len: length σs = length ts by auto
  from C-sub-S[OF Fun(1)] have inS: σ ∈ S set σs ⊆ S by auto
  hence σs: ⋀ j. j < length ts ⟹ σs ! j ∈ S using len unfolding set-conv-nth
by auto
  show ?case
  proof (rule list-all2-nthD[OF Fun(3) i, rule-format, OF x])
    show ¬ inf-sort (σs ! i) unfolding inf-sort[OF σs[OF i]] finite-sort-def
    proof
      assume inf: infinite {t. t : σs ! i in 𝒯(C)}
      {
        fix j
        assume j < length ts
        from σs[OF this] have σs ! j ∈ S by auto
        from sorts-non-empty[OF this] have ∃ tj. tj : σs ! j in 𝒯(C) by blast
      }
      hence ∀ j. ∃ tj. j < length ts ⟶ tj : σs ! j in 𝒯(C) by auto
      from choice[OF this] obtain tj where
        tj: j < length ts ⟹ tj j : σs ! j in 𝒯(C) for j by auto
      define ft where ft t = Fun f (map (tj (i := t)) [0..< length ts]) for t
      {
        fix t
        assume t : σs ! i in 𝒯(C)
        hence ft t : σ in 𝒯(C) unfolding ft-def using tj
          by (intro Fun-hastypeI[OF Fun(1)] list-all2-all-nthI, auto simp: len)
      } note ft = this
      have inj: inj ft unfolding ft-def using i by (auto simp: inj-def)
      from inf inj have infinite (ft ` {t. t : σs ! i in 𝒯(C)})
        by (metis finite-imageD inj-def inj-on-def)
      with ft have infinite {t. t : σ in 𝒯(C)}
        by (metis (no-types, lifting) finite-subset image-subset-iff mem-Collect-eq)
      with Fun(5) inf-sort[OF inS(1)]
      show False unfolding finite-sort-def by auto
    qed
  qed
qed auto


context
  fixes CC :: 'f × 's list ⇒ 's option
    and renVar :: 'v ⇒ 'v
    and renNat  :: nat ⇒ 'v
    and fidl-solver :: ((nat×'s) × int)list × - ⇒ bool
  assumes CC: improved ⟹ CC = C
```

and *renaming-ass*: *improved* $\Longrightarrow$ *renaming-funs renNat renVar*
and *fidl-solver*: *improved* $\Longrightarrow$ *finite-idl-solver fidl-solver*
begin

abbreviation *Match-decomp'-impl* **where** *Match-decomp'-impl* $\equiv$ *match-decomp'-impl renNat*
abbreviation *Decomp'-main-loop* **where** *Decomp'-main-loop* $\equiv$ *decomp'-main-loop renNat*
abbreviation *Decomp'-impl* **where** *Decomp'-impl* $\equiv$ *decomp'-impl renNat*
abbreviation *Pat-inner-impl* **where** *Pat-inner-impl* $\equiv$ *pat-inner-impl renNat*
abbreviation *Pat-impl* **where** *Pat-impl* $\equiv$ *pat-impl CC renNat*
abbreviation *Pats-impl* **where** *Pats-impl* $\equiv$ *pats-impl CC renNat fidl-solver*
abbreviation *Pat-complete-impl* **where** *Pat-complete-impl* $\equiv$ *pat-complete-impl CC renNat renVar fidl-solver*

definition *allowed-vars* **where** *allowed-vars n* = (*if improved then range renVar* $\cup$ *renNat* ‘ {..<*n*} *else UNIV*)

definition *lvar-cond* **where** *lvar-cond n V* = (*V* $\subseteq$ *allowed-vars n*)
definition *lvar-cond-mp* **where** *lvar-cond-mp n mp* = *lvar-cond n* (*lvars-mp mp*)
definition *lvar-cond-pp* **where** *lvar-cond-pp n pp* = *lvar-cond n* (*lvars-pp pp*)

lemma *lvar-cond-simps*[*simp*]:
lvar-cond n (*insert x A*) = (*x* $\in$ *allowed-vars n* $\wedge$ *lvar-cond n A*)
lvar-cond n {}
lvar-cond n (*A* $\cup$ *B*) = (*lvar-cond n A* $\wedge$ *lvar-cond n B*)
lvar-cond n ($\bigcup$ *As*) = ($\forall$ *A* $\in$ *As*. *lvar-cond n A*)
unfolding *lvar-cond-def* **by** *auto*

lemma *lvar-cond-mono*: *n* $\leq$ *n'* $\Longrightarrow$ *lvar-cond n V* $\Longrightarrow$ *lvar-cond n' V*
unfolding *lvar-cond-def allowed-vars-def* **by** (*auto split*: *if-splits*)

lemma *pair-fst-imageI*: (*a*,*b*) $\in$ *c* $\Longrightarrow$ *a* $\in$ *fst* ‘ *c* **by** *force*

lemma *not-in-fstD*: *x* $\notin$ *fst* ‘ *a* $\Longrightarrow$ $\forall$ *z*. (*x*,*z*) $\notin$ *a* **by** *force*

lemma *many-remdups-steps*: **assumes** *mp-mset mp2* = *mp-mset mp1 mp2* $\subseteq\#$ *mp1*
shows $(\rightarrow_m)^{**}$ *mp1 mp2*
proof −
from *assms* **obtain** *mp3* **where** *mp1*: *mp1* = *mp3* + *mp2*
by (*metis subset-mset.less-eqE union-commute*)
from *assms*(*1*)[*unfolded mp1*] **have** *mp-mset mp3* $\subseteq$ *mp-mset mp2* **by** *auto*
thus *?thesis* **unfolding** *mp1*
proof (*induct mp3*)
case (*add pair mp3*)
from *add* **have** *IH*: $(\rightarrow_m)^{**}$ (*mp3* + *mp2*) *mp2* **by** *auto*
from *add* **have** *pair* $\in\#$ *mp3* + *mp2* **by** *auto*

```
   then obtain mp4 where mp3 + mp2 = add-mset pair mp4 by (rule mset-add)
    from match-duplicate[of pair mp4, folded this] IH
    show ?case by simp
  qed auto
qed

lemma many-match-steps:
  assumes ⋀ t l. (t,l) ∈# mp1 ⟹ ∃ x. l = Var x ∧ x ∉ lvars-mp (mp1 − {#
(t,l) #} + mp2)
  shows (→m)** (mp1 + mp2) mp2
  using assms
proof (induct mp1)
  case (add pair mp1)
  obtain t l where pair: pair = (t, l) by force
  from add(2)[of t l, unfolded pair] obtain x where
    l: l = Var x and x: x ∉ lvars-mp (mp1 + mp2)
    by auto
  from match-match[of x mp1 + mp2 t, folded l, folded pair]
 have add-mset pair (mp1 + mp2) →m mp1 + mp2 using x unfolding lvars-mp-def
by auto
  also have (→m)** (mp1 + mp2) mp2
    by (rule add, insert add(2), force simp: lvars-mp-def)
  finally show ?case by simp
qed auto



lemma decomp'-impl: assumes
    wf-lr2 mp
    set xs = lvars-mp (mp-list (mp-lx (fst mp)))
    lvar-cond-mp n (mp-lr mp)
    Decomp'-impl n xs mp = (n',mp')
    improved
  shows wf-lr3 mp'
    lvar-cond-mp n' (mp-lr mp')
    (→m)** (mp-lr mp) (mp-lr mp')
    n ≤ n'
proof (atomize (full), goal-cases)
  case 1
  obtain xl rx b where mp: mp = (xl,rx,b) by (cases mp, auto)
  define out where out = ([] :: ('v,('f,nat × 's)term list) alist)
  let ?lr = λ rx. (xl,rx,b)
  define Measure where Measure (rx :: ('v,('f,nat × 's)term list) alist) =
    sum-list (map ((λ ts. sum-list (map size ts)) o snd) rx) for rx
  define cond3 where cond3 ts = (xl = [] ⟶ wf-ts3 ts) for ts
  {
    fix out rx'
    assume Decomp'-main-loop n xs rx out = (n', rx')
      wf-lr2 (?lr (rx @ out))
```

lvar-cond-mp n (mp-lr (?lr (rx @ out)))
Ball (snd ‘ set out) cond3
hence *wf-lr3 (?lr rx′) ∧ lvar-cond-mp n′ (mp-lr (?lr rx′))*
*∧ (→m)** (mp-lr (?lr (rx @ out))) (mp-lr (?lr rx′))*
∧ n ≤ n′
proof (*induct rx arbitrary*: *n n′ out rx′ rule*: *wf-induct*[*OF wf-measure*[*of Measure*]])
case (*1 rx n n′ out rx′*)
note *IH* = *1*(*1*)[*rule-format*]
have *Decomp′-main-loop n xs rx out* = (*n′*, *rx′*) **by** *fact*
note *res* = *this*[*unfolded decomp′-main-loop.simps*[*of - - - rx*]]
note *wf* = *1*(*3*)
note *lvc* = *1*(*4*)
note *cond3* = *1*(*5*)
show *?case*
proof (*cases rx*)
case *Nil*
from *Nil* **have** *mset*: *mset* (*rx* @ *out*) = *mset out* **by** *auto*
note *wf* = *wf*[*unfolded wf-lr2-mset*[*OF mset*]]
note *mp-lr* = *mp-lr-mset*[*OF mset*]
show *?thesis* **using** *Nil wf lvc res cond3* **unfolding** *mp-lr*
by (*auto simp*: *wf-lr3-def wf-lr2-def wf-rx3-def cond3-def*)
next
case (*Cons pair rx2*)
then obtain *x ts* **where** *rx*: *rx* = (*x*,*ts*) # *rx2* **by** (*cases pair*, *auto*)
let *?cond* = *tl ts* = [] ∨ (∃ *t* ∈ *set ts*. *is-Var t*) ∨ *x* ∈ *set xs*
note *res* = *res*[*unfolded rx split list.simps*]
from *wf*[*unfolded rx wf-lr2-def split*] **have** *wfts*: *wf-ts ts* ∨ *wf-ts2 ts*
and *dist-vars*: *distinct* (*x* # *map fst* (*rx2* @ *out*))
by (*auto simp*: *wf-rx-def wf-rx2-def split*: *if-splits*)
hence *ts*: *ts* ≠ [] **unfolding** *wf-ts-def wf-ts2-def* **by** *auto*
show *?thesis*
proof (*cases ?cond*)
case *True*
hence *?cond* = *True* **by** *simp*
note *res* = *res*[*unfolded this if-True*]
have *mset*: *mset* (*rx* @ *out*) = *mset* (*rx2* @ (*x*, *ts*) # *out*) **unfolding** *rx* **by** *auto*
note *wf* = *wf*[*unfolded wf-lr2-mset*[*OF mset*]]
note *mp-lr* = *mp-lr-mset*[*OF mset*]
have *c3*: *cond3 ts*
unfolding *cond3-def*
proof (*intro impI*)
assume *xl*: *xl* = []
with *assms*[*unfolded mp*] **have** *xs*: *xs* = [] **unfolding** *lvars-mp-def* **by** *auto*
from *wf*[*unfolded wf-lr2-def split*] *xl* **have** *wf-ts2 ts* **unfolding** *wf-rx2-def* **by** *auto*
hence *tl ts* ≠ [] **unfolding** *wf-ts2-def* **by** (*cases ts*, *auto*)

```
      with True xs have ∃ t∈set ts. is-Var t by auto
      thus wf-ts3 ts unfolding wf-ts3-def by auto
    qed
    have (rx2, rx) ∈ measure Measure unfolding Measure-def rx using ts
      by (cases ts, auto)
    note IH = IH[OF this res wf lvc[unfolded mp-lr], folded mp-lr]
    show ?thesis
      by (rule IH, insert c3 cond3, auto)
  next
    case False
    define l where l = num-args (hd ts)
    define k where k = length ts
    define fresh where fresh = map renNat [n..<n + l]
    define rx1 where rx1 = zipAll fresh (map args ts)
    from ts have 0: 0 < length ts and k0: k ≠ 0 by (auto simp: k-def)
    from ts have hd ts ∈ set ts by auto
    with False obtain f bs0 where hd: hd ts = Fun f bs0 by blast
     from False ts have k: k ≥ 2 unfolding k-def by (cases ts; cases tl ts;
auto)
    hence l0: l = length bs0 unfolding l-def using ts hd by auto
    from ts hd have ts0: ts ! 0 = Fun f bs0 by (cases ts, auto)
    from wfts[unfolded wf-ts-def wf-ts2-def]
    have dist-noconf: distinct ts ∧ (∀ j. 0 < j ⟶ j < length ts ⟶ conflicts
(ts ! 0) (ts ! j) ≠ None) by auto
    have lfresh: length fresh = l unfolding fresh-def by simp
  from renaming-ass[unfolded renaming-funs-def, rule-format, OF ‹improved›]

     have ren: inj renNat inj renVar range renNat ∩ range renVar = {} by
auto
    {
      fix t
      assume tts: t ∈ set ts
      from False tts obtain g bs where t: t = Fun g bs by (cases t, auto)
      with tts obtain i where i: i < length ts and tsi: ts ! i = Fun g bs
        unfolding set-conv-nth by auto
      have length bs = l ∧ g = f
      proof (cases i = 0)
        case True
        with ts0 l0 tsi show ?thesis by auto
      next
        case False
        with i dist-noconf have conflicts (ts ! 0) (ts ! i) ≠ None by auto
        from this[unfolded tsi ts0] l0 show ?thesis
          by (auto simp: conflicts.simps split: if-splits)
      qed
      with t tts have ∃ bs. t = Fun f bs ∧ length bs = l by auto
    } note no-conflict = this
    define t where t = (λ i j. args (ts ! i) ! j)
    have ts = map (λ i. ts ! i) [0..<k] unfolding k-def
```

```
    by (intro nth-equalityI, auto)
  also have ... = map (λ i. Fun f (map (t i) [0 ..< l])) [0..<k]
  proof (intro map-cong[OF refl])
    fix i
    assume i ∈ set [0..<k]
    hence ts ! i ∈ set ts unfolding k-def by auto
    from no-conflict[OF this] obtain bs where tsi: ts ! i = Fun f bs and
      len: length bs = l by auto
    show ts ! i = Fun f (map (t i) [0..<l]) unfolding tsi term.simps term.sel
t-def
      using len by (intro conjI nth-equalityI, auto)
  qed
  finally have ts-t: ts = map (λi. Fun f (map (t i) [0..<l])) [0..<k] .
  {
    fix bs
    assume bs ∈ set (map args ts)
    hence length bs = l using no-conflict by force
  }
  from zipAll[OF lfresh, of map args ts, OF this, unfolded map-map o-def,
folded rx1-def]
  have rx1 = map (λj. (fresh ! j, map (λbs. bs ! j) (map args ts))) [0..<l]
by auto
  also have ... = map (λi. (fresh ! i, map (λ j. t j i) [0..<k])) [0..<l]
    by (intro nth-equalityI, auto simp: ts-t)
   finally have rx1: rx1 = map (λi. (fresh ! i, map (λ j. t j i) [0..<k]))
[0..<l] .
  define rrx where rrx = map (λ(y, ts'). (y, remdups ts')) rx1
  define frrx where frrx = filter (λ(y, ts'). tl ts' ≠ []) rrx
  from False have ?cond = False by simp
  note res = res[unfolded this if-False, folded l-def,
      unfolded Let-def, folded fresh-def, folded rx1-def, folded rrx-def, folded
frrx-def]

  let ?meas = λ rx. sum-list (map ((λts. sum-list (map size ts)) ∘ snd) rx)
  have snd-case: snd (case x of (y :: 'v, ts') ⇒ (y, remdups ts')) = remdups
(snd x) for x by (cases x, auto)
  have fst-case: fst (case x of (y :: 'v, ts') ⇒ (y, remdups ts')) = fst x for x
by (cases x, auto)
  have sum-remdups: sum-list (map size (remdups b)) ≤ sum-list (map size
b) for b by (induct b, auto)
  have ?meas frrx ≤ ?meas rrx unfolding frrx-def by (induct rrx, auto)
  also have ... ≤ ?meas rx1 unfolding rrx-def
      by (induct rx1, auto simp: o-def split: prod.splits intro!: add-mono
sum-remdups)
  also have ... = (∑ x←[0..<l]. ∑ xa←[0..<k]. size (t xa x))
    unfolding rx1 map-map o-def snd-conv by simp
  also have ... = (∑ xa←[0..<k]. ∑ x←[0..<l]. size (t xa x))
    unfolding sum.list-conv-set-nth by (auto intro: sum.swap)
  also have ... < sum-list (map size ts)
```

unfolding *ts-t map-map o-def*
by (*intro sum-list-strict-mono*, *insert k0*, *auto simp*: *o-def size-list-conv-sum-list*)
finally have *measure*: (*frrx* @ *rx2*, *rx*) ∈ *measure Measure* **unfolding** *Measure-def rx*
by *simp*

have *left*: *mp-lr* (*xl*, *rx* @ *out*, *b*) = *mp-rx* ([(*x*,*ts*)],*b*) + *mp-lr* (*xl*, *rx2* @ *out*, *b*)
unfolding *mp-lr-def split mp-rx-def rx* **by** (*auto simp*: *List.maps-def*)
have *right*: *mp-lr* (*xl*, (*rx1* @ *rx2*) @ *out*, *b*) = *mp-rx* (*rx1* ,*b*) + *mp-lr* (*xl*, *rx2* @ *out*, *b*)
unfolding *mp-lr-def split mp-rx-def rx* **by** (*auto simp*: *List.maps-def*)
have *cong*: *mp0* + *mp2* →$_m$ *mp1* + *mp2* ⟹ *mp1* = *mp1'* ⟹ *mp0* + *mp2* →$_m$ *mp1'* + *mp2*
for *mp0 mp2 mp1 mp1'* :: (*'f*,*'v*,*'s*)*match-problem-mset* **by** *auto*
note *List.maps-def*[*simp*]
from *assms*(*2*)[*unfolded mp*]
have *xs*: *set xs* = *lvars-mp* (*mp-list* (*mp-lx xl*)) **by** *auto*
have *dist-fresh*: *distinct fresh* **unfolding** *fresh-def distinct-map*
using *ren* **by** (*auto simp*: *inj-def inj-on-def*)
have *lvars-fresh-disj*: *lvars-mp* (*mp-lr* (*xl*, *rx* @ *out*, *b*)) ∩ *set fresh* = {}
proof −
have *lvars-mp* (*mp-lr* (*xl*, *rx* @ *out*, *b*)) ⊆ *allowed-vars n*
using *1*(*4*) **unfolding** *lvar-cond-mp-def lvar-cond-def* **.**
moreover have *set fresh* ∩ *allowed-vars n* = {} **unfolding** *allowed-vars-def fresh-def*
using *ren*(*3*) ‹*improved*›
by (*auto dest*: *injD*[*OF ren*(*1*)])
ultimately show *?thesis* **by** *auto*
qed
have *step*: *mp-lr* (*xl*, *rx* @ *out*, *b*) →$_m$ *mp-lr* (*xl*, (*rx1* @ *rx2*) @ *out*, *b*)
unfolding *left right*
proof (*rule cong*[*OF match-decompose'*[*OF - - - lfresh - ‹improved›*, *of - x f*]])
show (*ti*, *y*) ∈# *mp-rx* ([(*x*, *ts*)], *b*) ⟹ *y* = *Var x* ∧ *root ti* = *Some* (*f*, *l*) **for** *ti y*
unfolding *ts-t mp-rx-def* **by** *auto*
from *False* **have** *xxs*: *x* ∉ *set xs* **by** *auto*
show (*ti*, *y*) ∈# *mp-lr* (*xl*, *rx2* @ *out*, *b*) ⟹ *x* ∉ *vars y* **for** *ti y*
using *dist-vars xxs*[*unfolded xs*]
by (*auto simp*: *mp-lr-def lvars-mp-def mp-rx-def dest*: *pair-fst-imageI*)

have *var-id*: ⋃ (*vars* ‘ *snd* ‘ *mp-mset* (*mp-rx* ([(*x*, *ts*)], *b*) + *mp-lr* (*xl*, *rx2* @ *out*, *b*)))
= *lvars-mp* (*mp-lr* (*xl*, *rx* @ *out*, *b*))
unfolding *rx lvars-mp-def mp-rx-def mp-lr-def split* **by** *auto*
show *lvars-disj-mp fresh* (*mp-mset* (*mp-rx* ([(*x*, *ts*)], *b*) + *mp-lr* (*xl*, *rx2* @ *out*, *b*)))

```
      unfolding lvars-disj-mp-def var-id
    proof
      show distinct fresh by fact
      have lvars-mp (mp-lr (xl, rx @ out, b)) ⊆ allowed-vars n
        using 1(4) unfolding lvar-cond-mp-def lvar-cond-def .
          moreover have set fresh ∩ allowed-vars n = {} unfolding al-
lowed-vars-def fresh-def
        using ren(3) ‹improved›
        by (auto dest: injD[OF ren(1)])
      ultimately show lvars-mp (mp-lr (xl, rx @ out, b)) ∩ set fresh = {}
by auto
    qed
    show 2 ≤ size (mp-rx ([(x, ts)], b))
      using k[unfolded k-def] unfolding mp-rx-def by auto
    define aux where aux i j = (t j i, Var (fresh ! i) :: ('f,'v)term) for i j
    have fresh-index: map Var fresh = map (λ i. Var (fresh ! i)) [0..<l]
      unfolding lfresh[symmetric]
      by (intro nth-equalityI, auto)
       have (∑(t, l)∈#mp-rx ([(x, ts)], b). mp-list (zip (args t) (map Var
fresh)))
      = mset (concat (map (λ t. zip (args t) (map Var fresh)) ts))
      unfolding mp-rx-def by (induct ts, auto)
     also have ... = mset (concat (map (λ j. map (λ i. (t j i,Var (fresh !
i))) [0..<l]) [0..<k]))
      unfolding ts-t map-map o-def
      apply (intro arg-cong[of - - mp-list])
      apply (intro arg-cong[of - - concat])
      apply (intro map-cong[OF refl])
      apply (subst zip-nth-conv)
      by (auto simp: fresh-def)
    also have ... = mp-rx (rx1, b)
      unfolding mp-rx-def
       by (auto simp add: rx1 o-def fresh-index ts-t mset-concat-union intro:
mset-sum-reindex)
    finally show (∑(t, l)∈#mp-rx ([(x, ts)], b). mp-list (zip (args t) (map
Var fresh))) =
      mp-rx (rx1, b) .
  qed

  have rrx-seteq: mp-mset (mp-rx (rrx, b)) = mp-mset (mp-rx (rx1, b))
 unfolding mp-rx-def rrx-def by (induct rx1, auto simp: o-def mset-concat)
 have glob-rrx-set-eq: mp-mset (mp-lr (xl, (rx1 @ rx2) @ out, b)) = mp-mset
(mp-lr (xl, (rrx @ rx2) @ out, b))
    unfolding mp-lr-def split mp-rx-append using rrx-seteq by auto
  have frrx-sub: mp-mset (mp-rx (frrx, b)) ⊆ mp-mset (mp-rx (rx1, b))
    unfolding rrx-seteq[symmetric]
 unfolding mp-rx-def frrx-def by (induct rrx, auto simp: o-def mset-concat)
 have glob-rrx-sub: mp-mset (mp-lr (xl, (frrx @ rx2) @ out, b)) ⊆ mp-mset
(mp-lr (xl, (rx1 @ rx2) @ out, b))
```

```
      unfolding mp-lr-def split mp-rx-append using frrx-sub by auto

    have lvc': lvar-cond-mp (n + l) (mp-lr (xl, (rx1 @ rx2) @ out, b))
      unfolding lvar-cond-mp-def lvar-cond-def
    proof
      fix y
      have rx': rx = [(x,ts)] @ rx2 unfolding rx by auto
      assume y ∈ lvars-mp (mp-lr (xl, (rx1 @ rx2) @ out, b))
       hence y ∈ lvars-mp (mp-lr (xl, rx @ out, b)) ∨ y ∈ lvars-mp (mp-rx
(rx1,b))
        unfolding rx'
        unfolding mp-lr-def split lvars-mp-def mp-rx-append by auto
      thus y ∈ allowed-vars (n + l)
      proof
        assume y ∈ lvars-mp (mp-lr (xl, rx @ out, b))
            with lvc have y ∈ allowed-vars n unfolding lvar-cond-mp-def
lvar-cond-def by auto
        thus ?thesis unfolding allowed-vars-def by auto
      next
        assume y ∈ lvars-mp (mp-rx (rx1, b))
      hence y ∈ set fresh unfolding rx1 lvars-mp-def mp-rx-def List.maps-def
         using lfresh by auto
        thus ?thesis unfolding fresh-def by (auto simp: allowed-vars-def)
      qed
    qed
    have lvars-mp (mp-lr (xl, (frrx @ rx2) @ out, b)) ⊆ lvars-mp (mp-lr (xl,
(rx1 @ rx2) @ out, b))
      using glob-rrx-sub
      unfolding lvars-mp-def by auto
     hence lvar-cond-new: lvar-cond-mp (n + l) (mp-lr (xl, (frrx @ rx2) @
out, b))
      using lvc' unfolding lvar-cond-mp-def lvar-cond-def by auto

    have wflx: wf-lx xl using wf unfolding wf-lr2-def by auto
    define ro where ro = rx @ out

    from wf[unfolded wf-lr2-def wf-rx2-def wf-rx-def]
    have dist-old: distinct (map fst (rx @ out)) by (auto split: if-splits)
    have dist-mid: distinct (map fst ((rx1 @ rx2) @ out))
    proof −
      from dist-old have distinct (map fst (rx2 @ out)) by (simp add: rx)
      moreover have set (map fst (rx2 @ out)) ∩ set fresh = {}
      proof (rule ccontr)
        assume ¬ ?thesis
        then obtain y where y: y ∈ set (map fst (rx2 @ out)) y ∈ set fresh
by auto
        from y obtain ts where yts: (y,ts) ∈ set (rx @ out) by (force simp:
rx)
        hence ts ∈ snd ‘ set (rx @ out) by force
```

```
          with wf[unfolded wf-lr-def wf-lr2-def wf-rx2-def wf-rx-def split fst-conv]
          have wf-ts ts ∨ wf-ts2 ts by metis
           with this[unfolded wf-ts-def wf-ts2-def] obtain t ts' where ts: ts = t
# ts' by (cases ts, auto)
          from yts[unfolded this] have y ∈ lvars-mp (mp-rx (rx @ out, b))
            unfolding lvars-mp-def mp-rx-def split fst-conv ro-def[symmetric]
            unfolding lvars-mp-def mp-lr-def mp-rx-def rx by force
          hence y ∈ lvars-mp (mp-lr (xl, rx @ out, b)) unfolding lvars-mp-def
mp-lr-def by auto
          with lvars-fresh-disj have y ∉ set fresh by auto
          with y show False by auto
        qed
        moreover have map fst rx1 = fresh unfolding rx1 using lfresh
          by (intro nth-equalityI, auto)
        ultimately show ?thesis using dist-fresh by auto
      qed
      also have map fst ((rx1 @ rx2) @ out) = map fst ((rrx @ rx2) @ out)
        unfolding rrx-def by auto
      finally have dist-new: distinct (map fst ((frrx @ rx2) @ out)) = True
        unfolding frrx-def by (auto simp: distinct-map-filter)


      from wf[unfolded rx wf-lr2-def split wf-rx2-def wf-rx-def fst-conv]
      have wf-ts: wf-ts ts ∨ wf-ts2 ts by (auto split: if-splits)
      {
        fix i j
        assume ij: i < k j < k
        from wf-ts[unfolded wf-ts-def wf-ts2-def] ij
       have conflicts (ts ! i) (ts ! j) ≠ None ∨ conflicts (ts ! j) (ts ! i) ≠ None
          unfolding k-def by (cases i < j; cases i = j; auto)
        with conflicts-sym have conflicts (ts ! i) (ts ! j) ≠ None
          by (metis rel-option-None2)
      } note ts-no-conflict = this
      let ?old = mp-rx (rx @ out, b)
      let ?mid = mp-rx ((rx1 @ rx2) @ out, b)
      let ?new = mp-rx ((frrx @ rx2) @ out, b)
      have mp-mset (mp-rx (frrx, b)) ≤ mp-mset (mp-rx (rrx, b))
        unfolding mp-rx-def frrx-def by auto
      also have rrx-rx1: ... ⊆ mp-mset (mp-rx (rx1, b))
        unfolding mp-rx-def rrx-def by auto
       finally have frrx-sub-rx1: mp-mset (mp-rx (frrx, b)) ⊆ mp-mset (mp-rx
(rx1, b)) .
    hence new-sub-mid: mp-mset ?new ⊆ mp-mset ?mid unfolding mp-rx-append
by auto

      have b-correct: (b = inf-var-conflict (mp-mset ?new)) = True
      proof −
        let ?old = mp-rx (rx @ out, b)
        let ?mid = mp-rx ((rx1 @ rx2) @ out, b)
```

let *?new* = *mp-rx* ((*frrx* @ *rx2*) @ *out*, *b*)
from *wf*[*unfolded wf-lr2-def wf-rx2-def wf-rx-def*]
have *b* = *inf-var-conflict* (*mp-mset ?old*) **by** (*auto split*: *if-splits*)
also have ... = *inf-var-conflict* (*mp-mset ?new*) (**is** *?inf-old* = *?inf-new*)
proof
assume *?inf-old*
from *this*[*unfolded inf-var-conflict-def*]
obtain *u w y z* **where**
u: (*u*, *Var y*) ∈# *?old* **and**
w: (*w*, *Var y*) ∈# *?old* **and**
conf: *Conflict-Var u w z* **and**
inf: *inf-sort* (*snd z*) **by** *auto*
show *?inf-new*
proof (*cases y* = *x*)
case *False*
hence (*u*, *Var y*) ∈# *?new* (*w*, *Var y*) ∈# *?new* **using** *u w*
unfolding *rx mp-rx-append mp-rx-Cons split* **by** *auto*
with *conf inf* **show** *?thesis* **unfolding** *inf-var-conflict-def* **by** *blast*
next
case *True*
with *dist-old u w* **have** *uw-ts*: *u* ∈ *set ts w* ∈ *set ts*
unfolding *rx mp-rx-Cons mp-rx-append split*
by (*auto simp*: *mp-rx-def dest*!: *not-in-fstD*)
with *conf* **have** *uw*: *u* ≠ *w* **by** *auto*
from *uw-ts*(*1*) **obtain** *i* **where** *i*: *i* < *k* **and** *u*: *u* = *ts* ! *i*
unfolding *k-def* **by** (*auto simp*: *set-conv-nth*)
from *uw-ts*(*2*) **obtain** *j* **where** *j*: *j* < *k* **and** *w*: *w* = *ts* ! *j*
unfolding *k-def* **by** (*auto simp*: *set-conv-nth*)
from *u w uw* **have** *ij*: *i* ≠ *j* **by** *auto*
have *id*: ((*f*, *length* [*0*..<*l*]) = (*f*, *length* [*0*..<*l*])) = *True* **by** *simp*
have *Conflict-Var* (*Fun f* (*map* (*t i*) [*0*..<*l*])) (*Fun f* (*map* (*t j*) [*0*..<*l*])) *z*
using *conf*[*unfolded u w ts-t*] *i j* **by** *auto*
note ∗ = *this*[*unfolded conflicts.simps length-map id if-True*]
from ∗ **obtain** *cs* **where** *those*: *those* (*map2 conflicts* (*map* (*t i*) [*0*..<*l*]) (*map* (*t j*) [*0*..<*l*])) = *Some cs* (**is** *?th* = -)
by (*cases ?th*, *auto*)
from ∗[*unfolded those*] **obtain** *c* **where** *c*: *c* ∈ *set cs* **and** *z*: *z* ∈ *set c* **by** *auto*
from *arg-cong*[*OF those*[*unfolded those-eq-Some*], *of length*]
have *lcs*: *length cs* = *l* **by** *auto*
with *c* **obtain** *a* **where** *a*: *a* < *l* **and** *c*: *c* = *cs* ! *a* **by** (*auto simp*: *set-conv-nth*)
from *arg-cong*[*OF those*[*unfolded those-eq-Some*], *of* λ *cs*. *cs* ! *a*]
have *conflicts* (*t i a*) (*t j a*) = *Some c* **using** *lcs a c* **by** *auto*
with *z* **have** *conf*: *Conflict-Var* (*t i a*) (*t j a*) *z* **by** *auto*
hence *diff*: *t i a* ≠ *t j a* **by** *auto*
let *?rd* = *remdups* (*map* (λ*j*. *t j a*) [*0*..<*k*])
from *i j* **have** *t i a* ∈ *set ?rd t j a* ∈ *set ?rd* **by** *auto*

```
            with diff have tl: tl (remdups (map (λj. t j a) [0..<k])) ≠ []
             by (cases remdups (map (λj. t j a) [0..<k]); cases tl (remdups (map
(λj. t j a) [0..<k])), auto)
             have mem: (t i a, Var (fresh ! a)) ∈# mp-rx (frrx,b) ∧ (t j a, Var
(fresh ! a)) ∈# mp-rx (frrx,b)
              unfolding frrx-def rrx-def rx1 using a i j tl
                unfolding mp-rx-def map-map o-def split fst-conv List.maps-def
in-multiset-in-set
              by (intro conjI, auto intro!: bexI[of - a])
            hence tia: (t i a, Var (fresh ! a)) ∈# ?new
             and tja: (t j a, Var (fresh ! a)) ∈# ?new unfolding mp-rx-append
by auto
           from tia tja conf inf show ?thesis unfolding inf-var-conflict-def by
blast
          qed
        next
          assume ?inf-new
          from this[unfolded inf-var-conflict-def]
          obtain u w y z where
            u: (u, Var y) ∈# ?new and
            w: (w, Var y) ∈# ?new and
            conf: Conflict-Var u w z and
            inf: inf-sort (snd z) by auto
          from u w new-sub-mid have u: (u, Var y) ∈# ?mid and w: (w, Var
y) ∈# ?mid by auto
          show ?inf-old
           proof (cases (u, Var y) ∈# mp-rx (rx2 @ out, b) ∧ (w, Var y) ∈#
mp-rx (rx2 @ out, b))
            case True
            hence (u, Var y) ∈# ?old (w, Var y) ∈# ?old using u w
              unfolding rx mp-rx-append mp-rx-Cons split by auto
            with conf inf show ?thesis unfolding inf-var-conflict-def by blast
          next
            case False
            then obtain v where (v, Var y) ∈# mp-rx (rx1, b) using u w
              unfolding mp-rx-append rx by auto
           hence y: y ∈ set (map fst rx1) y ∈ set fresh unfolding rx1 mp-rx-def
using lfresh by auto
            with dist-mid have yro: y ∉ fst ‘ set (rx2 @ out) by auto
            from not-in-fstD[OF yro] u
            have u: (u,Var y) ∈# mp-rx (rx1, b) unfolding mp-rx-def by auto
            from not-in-fstD[OF yro] w
           have w: (w,Var y) ∈# mp-rx (rx1, b) unfolding mp-rx-def by auto
            from y obtain a where a: a < l and y: y = fresh ! a using lfresh
by (auto simp: set-conv-nth)
            from u[unfolded mp-rx-def rx1] obtain a′ i
              where u: u = t i a′ i < k y = fresh ! a′ a′ < l
              by auto
           from y[unfolded u] have a′ = a unfolding fresh-def using ‹a′ < l› a
```

```
            by (auto dest: injD[OF ren(1)])
          note u = u(1−2)[unfolded this]
          from w[unfolded mp-rx-def rx1] obtain a' j
            where w: w = t j a' j < k y = fresh ! a'  a' < l
            by auto
        from y[unfolded w] have a' = a unfolding fresh-def using ‹a' < l› a
            by (auto dest: injD[OF ren(1)])
          note w = w(1−2)[unfolded this]
          from ts-no-conflict[OF u(2) w(2)] obtain cs where
            conf-ij: conflicts (ts ! i) (ts ! j) = Some cs by auto
        hence conflicts (Fun f (map (t i) [0..<l])) (Fun f (map (t j) [0..<l]))
= Some cs
            unfolding ts-t using u(2) w(2) by auto
          from this[unfolded conflicts.simps]
            have map-option concat (those (map2 conflicts (map (t i) [0..<l])
(map (t j) [0..<l]))) = Some cs
            by auto
             then obtain css where those: those (map2 conflicts (map (t i)
[0..<l]) (map (t j) [0..<l])) = Some css
            and cs: cs = concat css by force
          from conf[unfolded u w] obtain csi where
            conf: conflicts (t i a) (t j a) = Some csi and z: z ∈ set csi
            by auto
          from arg-cong[OF those[unfolded those-eq-Some], of length]
          have lcss: length css = l by auto
         from arg-cong[OF those[unfolded those-eq-Some], of λ xs. xs ! a] lcss
conf z
          have z ∈ set (css ! a) using a by simp
          with lcss a cs have z ∈ set cs by auto
          with conf-ij have Conflict-Var (ts ! i) (ts ! j) z by auto
          moreover have (ts ! i, Var x) ∈# ?old using u(2)
            unfolding rx mp-rx-Cons mp-rx-append split k-def by auto
          moreover have (ts ! j, Var x) ∈# ?old using w(2)
            unfolding rx mp-rx-Cons mp-rx-append split k-def by auto
           ultimately show ?thesis using inf unfolding inf-var-conflict-def
by blast
        qed
      qed
      finally show ?thesis
        by (simp add: mp-rx-append rrx-seteq)
    qed


    {
      fix y ts' t1 t2
      assume ∗: (y,ts') ∈ set ((rx1 @ rx2) @ out) t1 ∈ set ts' t2 ∈ set ts'
      have conflicts t1 t2 ≠ None
      proof
        assume conf: conflicts t1 t2 = None
```

```
            hence diff: t1 ≠ t2 by auto
             from conf have conf': conflicts t2 t1 = None using conflicts-sym[of
t1 t2] by auto
            from ∗(2−3) obtain i where t1: t1 = ts' ! i and i: i < length ts' by
(auto simp: set-conv-nth)
            from ∗(2−3) obtain j where t2: t2 = ts' ! j and j: j < length ts' by
(auto simp: set-conv-nth)
            from diff i j t1 conf conf' obtain i j where
              ij: j < length ts' i < j and
              conf: conflicts (ts' ! i) (ts' ! j) = None
              unfolding t1 t2
              by (cases i < j; cases j < i; auto)
            show False
            proof (cases (y,ts') ∈ set rx1)
              case False
              hence (y,ts') ∈ set (rx @ out) using ∗ unfolding rx by auto
              with wf[unfolded wf-lr2-def wf-rx2-def wf-rx-def]
                have wf-ts: wf-ts ts' ∨ wf-ts2 ts' unfolding ro-def[symmetric] by
(auto split: if-splits)
              with ij conf show False unfolding wf-ts-def wf-ts2-def by blast
            next
              case True
              from this[unfolded rx1] obtain a where a: a < l
                and ts': ts' = map (λj. t j a) [0..<k]
                and lts': length ts' = k by auto
              from conf have conf: conflicts (t i a) (t j a) = None
                unfolding ts' using lts' a ij by auto
              from ij have ij: i < k j < k using lts' by auto
              have conf: conflicts (ts ! i) (ts ! j) = None
              unfolding ts-t using ij conf a by (force simp: conflicts.simps set-zip)
              with ts-no-conflict[OF ij] show False ..
            qed
          qed
        } note no-clashes = this

        have True-id: (True ∧ b ∧ True) = b for b by simp
        have if-id: (if xl = [] then Ball P wf-ts2 else Ball P wf-ts)
          = Ball P (λ ts. if xl = [] then wf-ts2 ts else wf-ts ts) for P by auto


        have wf': wf-lr2 (xl, (frrx @ rx2) @ out, b)
          unfolding wf-lr2-def split wf-rx2-def wf-rx-def snd-conv fst-conv
          unfolding dist-new b-correct True-id if-id
        proof (intro conjI wflx ballI)
          fix ts'
          assume ts' ∈ snd ' set ((frrx @ rx2) @ out)
           then obtain y where (y,ts') ∈ set frrx ∨ ts' ∈ snd ' set (rx2 @ out)
by force
          thus if xl = [] then wf-ts2 ts' else wf-ts ts'
```

```
          proof
            assume ts' ∈ snd ` set (rx2 @ out)
           with wf show ?thesis unfolding wf-lr2-def split rx wf-rx2-def wf-rx-def
              by auto
          next
            assume (y,ts') ∈ set frrx
            from this[unfolded frrx-def]
            have tl: tl ts' ≠ [] and in-rrx: (y,ts') ∈ set rrx by auto
            from in-rrx[unfolded rrx-def] obtain ts'' where
              in-rx1: (y,ts'') ∈ set rx1 and
              rd: ts' = remdups ts'' by auto
           from in-rx1[unfolded rx1] have length ts'' = length ts unfolding k-def
by auto
            with ts have ts'': ts'' ≠ [] by auto
            with rd have ts': ts' ≠ [] by auto
            with tl have len2: length ts' ≥ 2 by (cases ts'; cases tl ts', auto)
            from rd have dist: distinct ts' by auto
            {
              fix j i
              assume j<length ts' i<j
              hence ts' ! i ∈ set ts' ts' ! j ∈ set ts' by auto
              hence ∗: ts' ! i ∈ set ts'' ts' ! j ∈ set ts'' unfolding rd by auto
              have conflicts (ts' ! i) (ts' ! j) ≠ None
                by (rule no-clashes[OF - ∗, of y], insert in-rx1, auto)
            }
           thus ?thesis unfolding wf-ts2-def wf-ts-def  using ts' len2 dist by auto
          qed
        qed


        note IH = IH[OF measure res wf' lvar-cond-new cond3[rule-format]]

        show ?thesis
        proof (intro conjI)
          show wf-lr3 (xl, rx', b) using IH by auto
          show lvar-cond-mp n' (mp-lr (xl, rx', b)) using IH by auto
          show n ≤ n' using IH by auto


           have mp-lr (xl, rx @ out, b) →m mp-lr (xl, (rx1 @ rx2) @ out, b) by
fact
         also have (→m)** (mp-lr (xl, (rx1 @ rx2) @ out, b)) (mp-lr (xl, (rrx @
rx2) @ out, b))
          proof (rule many-remdups-steps[OF glob-rrx-set-eq[symmetric]])
            have mp-rx (rrx, b) ⊆# mp-rx (rx1, b)
              unfolding rrx-def mp-rx-def fst-conv
            proof (induct rx1)
              case (Cons pair rx2)
              obtain x ts where pair: pair = (x,ts) by force
```

show *?case* **unfolding** *List.maps-simps list.simps pair split mset-append*

proof (*rule subset-mset.add-mono*[*OF - Cons*])
show *mp-list* (*map* (λ*t*. (*t*, *Var x*)) (*remdups ts*)) ⊆# *mp-list* (*map* (λ*t*. (*t*, *Var x*)) *ts*)
unfolding *mset-map*
by (*intro image-mset-subseteq-mono mset-remdups-subset-eq*)
qed
qed *auto*
thus *mp-lr* (*xl*, (*rrx* @ *rx2*) @ *out*, *b*) ⊆# *mp-lr* (*xl*, (*rx1* @ *rx2*) @ *out*, *b*)
unfolding *mp-lr-def split mp-rx-append* **by** *auto*
qed

also have (→$_m$)** (*mp-lr* (*xl*, (*rrx* @ *rx2*) @ *out*, *b*)) (*mp-lr* (*xl*, (*frrx* @ *rx2*) @ *out*, *b*))
proof −
define *long* :: ('*v* × ('*f*, *nat* × '*s*) *Term.term list*) ⇒ *bool* **where**
long = (λ(*y*, *ts'*). *tl ts'* ≠ [])
define *short* **where** *short* = *Not o long*
have *short-long*: *mp-rx* (*rrx*,*b*) = *mp-rx* (*filter short rrx*,*b*) + *mp-rx* (*filter long rrx*,*b*)
unfolding *mp-rx-def fst-conv short-def* **by** (*induct rrx*, *auto*)
hence *expand*: *mp-lr* (*xl*, (*rrx* @ *rx2*) @ *out*, *b*) =
mp-rx (*filter short rrx*, *b*) + *mp-lr* (*xl*, (*frrx* @ *rx2*) @ *out*, *b*)
unfolding *mp-lr-def split mp-rx-append short-long long-def frrx-def* **by** *simp*
show *?thesis* **unfolding** *expand*
proof (*rule many-match-steps*)
fix *s lhs*
assume (*s*, *lhs*) ∈# *mp-rx* (*filter short rrx*, *b*)
from *this*[*unfolded mp-rx-def fst-conv short-def long-def List.maps-def*, *simplified*]
obtain *y ts'* **where** *in-rrx*: (*y*, *ts'*) ∈ *set rrx* **and** *lhs*: *lhs* = *Var y*
and *tl ts'* = [] *s* ∈ *set ts'*
by *auto*
then have *ts'*: *ts'* = [*s*] **by** (*cases ts'*; *cases tl ts'*; *auto*)
show ∃ *x*. *lhs* = *Var x* ∧ *x* ∉ *lvars-mp*
(*mp-rx* (*filter short rrx*, *b*) − {#(*s*, *lhs*)#} + *mp-lr* (*xl*, (*frrx* @ *rx2*) @ *out*, *b*))
proof (*intro exI*[*of - y*] *conjI lhs notI*)
assume *mem*: *y* ∈ *lvars-mp* (*mp-rx* (*filter short rrx*, *b*) − {#(*s*, *lhs*)#} + *mp-lr* (*xl*, (*frrx* @ *rx2*) @ *out*, *b*))
from *in-rrx* **obtain** *a*
where *a*: *a* < *l* **and** *y*: *y* = *fresh* ! *a* **and** *yts'*: (*y*,*ts'*) = *rrx* ! *a*
unfolding *rrx-def rx1* **by** *auto*
with *lfresh* **have** *yfresh*: *y* ∈ *set fresh* **by** *auto*
with *lvars-fresh-disj mem*
have *y* ∈ *lvars-mp* (*mp-rx* (*filter short rrx*, *b*) − {#(*s*, *lhs*)#}) ∨ *y*

```
∈ lvars-mp (mp-rx (frrx, b))
              unfolding rx mp-lr-def split mp-rx-append mp-rx-Cons lvars-mp-def
by auto
              hence ∃ b. b < l ∧ y = fresh ! b ∧ a ≠ b
              proof
                assume y ∈ lvars-mp (mp-rx (frrx, b))
                  from this[unfolded frrx-def, folded long-def, unfolded mp-rx-def
lvars-mp-def, simplified]
                obtain ts'' where ts'': (y, ts'') ∈ set rrx and long: long (y, ts'')
by auto
                from long ts' have diff: ts' ≠ ts'' unfolding long-def by auto
                 from ts'' obtain b where b: b < l and yts'': (y, ts'') = rrx ! b
and y: y = fresh ! b
                  unfolding rrx-def rx1 by auto
                from yts' yts'' diff have diff: a ≠ b
                  by (metis snd-conv)
                with a b y show ?thesis by auto
              next
                assume mem: y ∈ lvars-mp (mp-rx (filter short rrx, b) − {#(s,
lhs)#})
                define other where other = take a rrx @ drop (Suc a) rrx
                have lenrrx: length rrx = l unfolding rrx-def rx1 by auto
                hence rrx = take a rrx @ rrx ! a # drop (Suc a) rrx using a
                  by (meson id-take-nth-drop)
                hence filter short rrx = filter short (take a rrx @ rrx ! a # drop
(Suc a) rrx) by simp
                 also have ... = filter short (take a rrx) @ rrx ! a # filter short
(drop (Suc a) rrx)
                   (is - = ?f1 @ - # ?f2)
                   by (simp add: yts'[symmetric] short-def long-def ts')
               also have rrx ! a = (y, [s]) unfolding yts'[symmetric] ts' by simp
                 also have mp-rx (?f1 @ ... # ?f2, b) − {#(s,lhs)#} = mp-rx
(?f1 @ ?f2,b)
                   unfolding mp-rx-append mp-rx-Cons lhs split by auto
                finally have y ∈ lvars-mp (mp-rx (?f1 @ ?f2,b)) using mem by
auto
                   from this[unfolded lvars-mp-def mp-rx-def, folded filter-append,
folded other-def]
                obtain ts'' where (y,ts'') ∈ set other by auto
               also have ... ⊆ {rrx ! b | b. b ∈ {..<length rrx} − {a}} unfolding
other-def
                   using a unfolding lenrrx[symmetric] unfolding set-conv-nth
                   by (auto simp: nth-append)
                    (metis (no-types, lifting) Suc-diff-diff diff-self-eq-0 diff-zero lessI
less-nat-zero-code neq0-conv
                        zero-less-diff)
                finally obtain b where b < l a ≠ b and (y, ts'') = rrx ! b using
lenrrx by auto
                then show ?thesis using lenrrx unfolding rrx-def rx1 by auto
```

```
            qed
            then obtain b where b < l y = fresh ! b a ≠ b by auto
              with y a show False using injD[OF ren(1), of n + a n + b]
unfolding fresh-def
              by auto
          qed
        qed
      qed
       also have (→m)** (mp-lr (xl, (frrx @ rx2) @ out, b)) (mp-lr (xl, rx',
b)) using IH by auto
       finally show (→m)** (mp-lr (xl, rx @ out, b)) (mp-lr (xl, rx', b)) .
     qed
   qed
  qed
 qed
 } note main = this
 from assms(4)[unfolded decomp'-impl-def mp split]
 obtain rx' where decomp: Decomp'-main-loop n xs rx [] = (n', rx') (is ?e = -)
  and mp': mp' = (xl, rx', b) by (cases ?e, auto)
 from main[OF decomp, unfolded append-Nil2, folded mp mp'] 1
 show ?case using assms by auto
qed

lemma match-decomp'-impl: assumes Match-decomp'-impl n mp = res
 and lvc: lvar-cond-mp n (mp-list mp)
 shows res = Some (n',mp') ⟹ (→m)** (mp-list mp) (mp-lr mp') ∧ wf-lr3 mp'
∧ lvar-cond-mp n' (mp-lr mp') ∧ n ≤ n'
   and res = None ⟹ ∃ mp'. (→m)** (mp-list mp) mp' ∧ match-fail mp'
proof (atomize (full), goal-cases)
 case 1
 note res = assms(1)[unfolded match-decomp'-impl-def]
 show ?case
 proof (cases match-steps-impl mp = None)
  case None: True
  with match-steps-impl(2)[OF refl None]
  show ?thesis using res by auto
 next
  case False
   then obtain xs mp2 where Some: match-steps-impl mp = Some (xs, mp2)
by auto
  note match = match-steps-impl(1)[OF refl Some]
  from lvc match have lvc: lvar-cond-mp n (mp-lr mp2)
   unfolding lvar-cond-def lvar-cond-mp-def by auto
  note res = res[unfolded Some option.simps split]
  show ?thesis
  proof (cases apply-decompose' mp2)
   case False
   obtain xl xr b where mp2: mp2 = (xl,xr,b) by (cases mp2, auto)
   from False[unfolded apply-decompose'-def mp2 split]
```

```
    have cond: improved ⟹ b ∨ xl ≠ [] by auto
    from match have wf-lr2 mp2 by simp
    with cond have wf-lr3 mp2
      unfolding wf-lr3-def wf-lr2-def mp2 split
      unfolding wf-rx3-def by auto
    with False res lvc match show ?thesis by auto
  next
    case True
    with res have res: res = Some (decomp'-impl renNat n xs mp2) by auto
    obtain n3 mp3 where dec: decomp'-impl renNat n xs mp2 = (n3, mp3) (is
?e = -) by (cases ?e) auto
    from True have improved unfolding apply-decompose'-def by (cases mp2,
auto)
    from match
    have steps12: (→m)** (mp-list mp) (mp-lr mp2)
      and wf2: wf-lr2 mp2
      and xs: set xs = lvars-mp (mp-list (mp-lx (fst mp2))) by auto
    from decomp'-impl[OF wf2 xs lvc dec ‹improved›] steps12
    show ?thesis unfolding res dec by auto
  qed
 qed
qed

lemma pat-inner-impl: assumes Pat-inner-impl n p pd = res
 and wf-pat-lr pd
 and tvars-pat (pat-mset (pat-mset-list p + pat-lr pd)) ⊆ V
 and lvar-cond-pp n (pat-mset-list p + pat-lr pd)
 shows res = None ⟹ (add-mset (pat-mset-list p + pat-lr pd) P, P) ∈ ⇛+
  and res = Some (n',p') ⟹ (add-mset (pat-mset-list p + pat-lr pd) P, add-mset
(pat-lr p') P) ∈ ⇛*
          ∧ wf-pat-lr p' ∧ tvars-pat (pat-mset (pat-lr p')) ⊆ V ∧ lvar-cond-pp n'
(pat-lr p') ∧ n ≤ n'
proof (atomize(full), insert assms, induct p arbitrary: n pd res n' 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 n pd res n'' 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 unfold-
ing pat-mset-list-def by auto
 from Cons(5) have lmp: lvar-cond-mp n (mp-list mp) unfolding lvar-cond-pp-def
lvar-cond-mp-def lvars-pp-def
   by (simp add: id)
 show ?case
 proof (cases Match-decomp'-impl n mp)
   case (Some pair)
  then obtain n' mp' where Some: Match-decomp'-impl n mp = Some (n', mp')
by (cases pair, auto)
   from match-decomp'-impl(1)[OF Some lmp refl]
```

have *steps*: $(\rightarrow_m)^{**}$ (*mp-list mp*) (*mp-lr mp'*) **and** *wf*: *wf-lr3 mp'*
and *lmp'*: *lvar-cond-mp n'* (*mp-lr mp'*) **and** *nn'*: $n \leq n'$ **by** *auto*
from *Cons*(*5*) *lvar-cond-mono*[*OF nn'*]
have *lvars-n'*: *lvar-cond-pp n'* (*pat-mset-list* (*mp # p*) + *pat-lr pd*)
by (*auto simp*: *lvar-cond-pp-def*)
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-pat* (*pat-mset* (*pat-mset-list p* + *pat-lr* (*mp' # pd*))) $\subseteq$ *V*
unfolding *id2* **by** (*auto simp*: *tvars-pat-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 split*]
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 *lmp' lvars-n'*
have *lvars-pre*: *lvar-cond-pp n'* (*pat-mset-list p* + *pat-lr* (*mp' # pd*))
unfolding *lvar-cond-pp-def lvar-cond-mp-def*
by (*auto simp*: *pat-mset-list-def lvars-pp-def lvars-mp-def pat-lr-def*)
from *res False* **have** *Pat-inner-impl n' p* (*mp' # pd*) = *res* **by** *auto*
from *Cons*(*1*)[*OF this wf vars lvars-pre*, *of n'' p'*, *unfolded id2*] *steps nn'*
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-decomp'-impl*(*2*)[*OF None lmp 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*) $\in$ *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*) $\in$ *P-step*$\widehat{\ }*$ **by** *auto*
note *res* = *Cons*(*2*)[*unfolded pat-inner-impl.simps None option.simps*]
have *vars*: *tvars-pat* (*pat-mset* (*pat-mset-list p* + *pat-lr pd*)) $\subseteq$ *V*
using *Cons*(*4*) **unfolding** *tvars-pat-def pat-mset-list-def* **by** *auto*
have *lvars*: *lvar-cond-pp n* (*pat-mset-list p* + *pat-lr pd*)
using *Cons*(*5*) **unfolding** *lvar-cond-pp-def lvars-pp-def* **by** (*auto simp*:

pat-mset-list-def)
  from *Cons*(*1*)[*OF res Cons*(*3*) *vars lvars*, *of n'' p'*] *steps*
  show *?thesis* **by** *auto*
 qed
qed

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

lemma *pat-impl*:
 assumes *Pat-impl n nl p* = *res*
  and *vars*: *tvars-pat* (*pat-list p*) ⊆ {..<*n*} × *S*
  and *lvarsAll*: ∀ *pp* ∈# *add-mset* (*pat-mset-list p*) *P*. *lvar-cond-pp nl pp*
 shows *res* = *Incomplete* ⟹ ∃ *p'*. (*add-mset* (*pat-mset-list p*) *P*, *add-mset p' P*) ∈ ⇛* ∧ *pat-fail p'*
  and *res* = *New-Problems* (*n'*,*nl'*,*ps*) ⟹ (*add-mset* (*pat-mset-list p*) *P*, *mset* (*map pat-mset-list ps*) + *P*) ∈ ⇛⁺
    ∧ *tvars-pat* (⋃ (*pat-list* ' *set ps*)) ⊆ {..< *n'*} × *S*
    ∧ (∀ *pp* ∈# *mset* (*map pat-mset-list ps*) + *P*. *lvar-cond-pp nl' pp*) ∧ *n* ≤ *n'*
  and *res* = *Fin-Var-Form fvf* ⟹ *improved*
    ∧ (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*pat-mset-list* (*pat-of-var-form-list fvf*)) *P*) ∈ ⇛*
    ∧ *finite-var-form-pat C* (*pat-list* (*pat-of-var-form-list fvf*))
    ∧ *Ball* (*set fvf*) (*distinct o map fst*)
    ∧ *Ball* (*set* (*concat fvf*)) (*distinct* ∘ *snd*)
proof (*atomize*(*full*), *goal-cases*)
 case *1*
 have *wf*: *wf-pat-lr* [] **unfolding** *wf-pat-lr-def* **by** *auto*
 have *vars*: *tvars-pat* (*pat-mset* (*pat-mset-list p*)) ⊆ {..<*n*} × *S*
  using *vars* **unfolding** *pat-mset-list* **by** *auto*
 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*]
 from *lvarsAll* **have** *lvars*: *lvar-cond-pp nl* (*pat-mset-list p*) **by** *auto*
 note *res* = *assms*(*1*)[*unfolded pat-impl-def*]
 show *?case*
 proof (*cases Pat-inner-impl nl p* [])
  case *None*
  from *pat-inner*(*1*)[*OF lvars this*] *res*[*unfolded None option.simps*] *vars*
  show *?thesis* **using** *lvarsAll* **by** (*auto simp*: *tvars-pat-def*)
 next
  case (*Some pair*)
  then obtain *nl'' p'* **where** *Some*: *Pat-inner-impl nl p* [] = *Some* (*nl''*, *p'*) **by** *force*
  from *pat-inner*(*2*)[*OF lvars Some*]
  have *steps*: (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*pat-lr p'*) *P*) ∈ ⇛*
   and *wf*: *wf-pat-lr p'*
   and *varsp'*: *tvars-pat* (*pat-mset* (*pat-lr p'*)) ⊆ {..<*n*} × *S*
   and *lvar-p'*: *lvar-cond-pp nl''* (*pat-lr p'*) **and** *nl*: *nl* ≤ *nl''* **by** *auto*
  obtain *ivc no-ivc* **where** *part*: *partition* (λ*mp*. *snd* (*snd mp*)) *p'* = (*ivc*, *no-ivc*) **by** *force*

from *part* **have** *no-ivc-filter*: *no-ivc* = *filter* (λ *mp*. ¬ (*snd* (*snd mp*))) *p′* **unfolding** *partition-filter-conv*
by (*auto simp*: *o-def*)
from *part* **have** *ivc-filter*: *ivc* = *filter* (λ *mp*. *snd* (*snd mp*)) *p′* **unfolding** *partition-filter-conv*
by (*auto simp*: *o-def*)
define *f* **where** *f* = (λ *mp* :: ('*f*,'*v*,'*s*)*match-problem-lr*. *snd* (*snd mp*))
from *part* **have** *Notf*: *no-ivc* = *filter* (*Not o f*) *p′* **unfolding** *partition-filter-conv f-def*
by (*auto simp*: *o-def*)
from *part* **have** *f*: *ivc* = *filter f p′* **unfolding** *partition-filter-conv f-def*
by (*auto simp*: *o-def*)
note *res* = *res*[*unfolded Some option.simps split part*]
show *?thesis*
proof (*cases* ∀ *mp*∈*set p′*. *snd* (*snd mp*))
case *True*
with *res part* **have** *res*: *res* = *Incomplete* **by** *auto*
have (*add-mset* (*pat-lr p′*) *P*, *add-mset* {#} *P*) ∈ ⇛*
proof (*cases pat-lr p′* = {#})
case *False*
have *add-mset* (*pat-lr p′* + {#}) *P* ⇛*m* {# {#} #} + *P*
proof (*intro P-simp-pp*[*OF pat-inf-var-conflict*[*OF - False*]] *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-lr3-def mp split*]
have *inf-var-conflict* (*set-mset* (*mp-rx* (*rx*,*b*))) **unfolding** *wf-rx-def wf-rx2-def wf-rx3-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 (*auto simp*: *tvars-pat-def*)
thus *?thesis* **unfolding** *P-step-def* **by** *auto*
qed *auto*
with *steps* **have** (*add-mset* (*pat-mset-list p*) *P*, *add-mset* {#} *P*) ∈ ⇛* **by** *auto*
moreover have *pat-fail* {#} **by** (*intro pat-empty*)
ultimately show *?thesis* **using** *res* **by** *auto*
next
case *False*
with *part* **have** *no-ivc*: *no-ivc* ≠ [] **unfolding** *partition-filter-conv o-def*
by (*metis* (*no-types*, *lifting*) *empty-filter-conv snd-conv*)
hence (*no-ivc* = []) = *False* **by** *auto*
note *res* = *res*[*unfolded this if-False*]
from *part* **have** *sub*: *set no-ivc* ⊆ *set p′ set ivc* ⊆ *set p′* **unfolding** *partition-filter-conv* **by** *auto*

```
{
  fix mp
  assume mp: mp ∈ set no-ivc
  with no-ivc-filter have b: ¬ snd (snd mp) by simp
  from mp sub have mp ∈ set p' by auto
  with wf[unfolded wf-pat-lr-def] have wf-lr3 mp by auto
  from this[unfolded wf-lr3-def wf-rx3-def wf-rx-def wf-rx2-def] b
  have ¬ inf-var-conflict (mp-mset (mp-rx (snd mp)))
    by (cases mp, auto split: if-splits)
  note b this
} note no-ivc-b = this

{
  fix mp
  assume mp: mp ∈ set ivc
  with ivc-filter have b: snd (snd mp) by simp
  from mp sub have mp ∈ set p' by auto
  with wf[unfolded wf-pat-lr-def] have wf-lr3 mp by auto
  from this[unfolded wf-lr3-def wf-rx3-def wf-rx-def wf-rx2-def] b
  have inf-var-conflict (mp-mset (mp-rx (snd mp)))
    by (cases mp, auto split: if-splits)
  note b this
} note ivc-b = this

define p'l where p'l = map mp-lr-list p'
let ?ivc'-cond = improved ∧ ivc ≠ [] ∧ (∀ mp∈set no-ivc. fst mp = [])
show ?thesis
proof (cases ?ivc'-cond)
  case True
  hence ?ivc'-cond = True by auto
  note res = res[unfolded this if-True, symmetric]
  from True CC have CC = C by auto
  note res = res[unfolded this]
  define M where M = pat-lr ivc
 let ?f = (λmp. ∀ xts∈set (fst (snd mp)). is-singleton-list (map T(C,V) (snd
xts)))
  define P' where P' = filter ?f no-ivc
  have P': set P' ⊆ set p' unfolding P'-def no-ivc-filter by auto
  have p'-split: pat-lr p' = M + pat-lr no-ivc
    unfolding pat-lr-def ivc-filter no-ivc-filter mset-map M-def
    by (induct p', auto)
  from no-ivc-filter have set no-ivc ⊆ set p' by auto
  hence steps2: (add-mset (M + pat-lr no-ivc) P, add-mset (M + pat-lr P')
P) ∈ ⇛* unfolding P'-def
  proof (induct no-ivc arbitrary: M)
    case (Cons mp mps M)
    show ?case
    proof (cases ?f mp)
      case True
```

have *add-mset* (*M* + *pat-lr* (*mp* # *mps*)) *P* = *add-mset* ((*M* + *pat-lr* [*mp*]) + *pat-lr mps*) *P*
unfolding *pat-lr-def* **by** *auto*
also have (..., *add-mset* ((*M* + *pat-lr* [*mp*]) + *pat-lr* (*filter ?f mps*)) *P*) ∈ ⇛*
by (*rule Cons*(*1*), *insert Cons*, *auto*)
also have (*M* + *pat-lr* [*mp*]) + *pat-lr* (*filter ?f mps*) = *M* + *pat-lr* (*filter ?f* (*mp* # *mps*))
unfolding *pat-lr-def* **using** *True* **by** *auto*
finally show *?thesis* **.**
next
case *False*
have *add-mset* (*M* + *pat-lr* (*mp* # *mps*)) *P* = *add-mset* (*add-mset* (*mp-lr mp*) (*M* + *pat-lr mps*)) *P*
unfolding *pat-lr-def* **by** *simp*
also have (..., {# *M* + *pat-lr mps* #} + *P*) ∈ ⇛ **unfolding** *P-step-def*
proof (*standard*, *unfold split*, *rule P-simp-pp*, *rule pat-remove-mp*)
obtain *xl xr b* **where** *mp*: *mp* = (*xl*,*xr*,*b*) **by** (*cases mp*, *auto*)
with *Cons*(*2*) **have** *mem*: (*xl*,*xr*,*b*) ∈ *set p′* **by** *auto*
from *mp False* **obtain** *x ts* **where** *xts*: (*x*,*ts*) ∈ *set xr*
and *nsingle*: ¬ *is-singleton-list* (*map* 𝒯(*C*,𝒱) *ts*) **by** *auto*
from *wf*[*unfolded wf-pat-lr-def*, *rule-format*, *OF mem*]
have *wf-lr3* (*xl*,*xr*,*b*) **by** *auto*
from *this*[*unfolded wf-lr3-def split*] **have** *wf-rx3* (*xr*,*b*) ∨ *wf-rx* (*xr*,*b*) **by** (*auto split*: *if-splits*)
with *xts* **have** *wf-ts2 ts* ∨ *wf-ts ts* **unfolding** *wf-rx3-def wf-rx2-def wf-rx-def*
by *auto*
hence *ts* ≠ [] **unfolding** *wf-ts2-def wf-ts-def* **by** *auto*
then obtain *t ts′* **where** *ts*: *ts* = *t* # *ts′* **by** (*cases ts*, *auto*)
from *nsingle*[*unfolded is-singleton-list ts singleton-def*]
obtain *t′* **where** *t′*: *t′* ∈ *set ts′* **and** *diff*: 𝒯(*C*,𝒱) *t* ≠ 𝒯(*C*,𝒱) *t′* **by** *force*
from *split-list*[*OF t′*] **obtain** *bef aft* **where** *ts′*: *ts′* = *bef* @ *t′* # *aft* **by** *auto*
from *split-list*[*OF xts*] **obtain** *bef′ aft′* **where** *xr*: *xr* = *bef′* @ (*x*, *ts*) # *aft′* **by** *auto*
obtain *M′* **where** *mp*: *mp-lr mp* = *add-mset* (*t*,*Var x*) (*add-mset* (*t′*, *Var x*) *M′*)
unfolding *mp ts ts′ xr*
unfolding *mp-lr-def* **by** (*auto simp*: *mp-rx-def List.maps-def*)
show *match-fail* (*mp-lr mp*) **unfolding** *mp*
by (*rule match-clash-sort*[*OF diff*])
qed
also have {# *M* + *pat-lr mps* #} + *P* = *add-mset* (*M* + *pat-lr mps*) *P* **by** *auto*
also have (..., *add-mset* (*M* + *pat-lr* (*filter ?f mps*)) *P*) ∈ ⇛*
by (*rule Cons*(*1*), *insert Cons*, *auto*)
also have *M* + *pat-lr* (*filter ?f mps*) = *M* + *pat-lr* (*filter ?f* (*mp* #

mps))
using *False* **by** *auto*
finally show *?thesis* **.**
qed
qed *auto*
from *steps*[*unfolded p'-split*] *steps2*
have *steps*: (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*M* + *pat-lr P'*) *P*) ∈ ⇛* **by** *auto*
have *step*: (*add-mset* (*M* + *pat-lr P'*) *P*, {# *pat-lr P'* #} + *P*) ∈ ⇛
unfolding *P-step-def*
proof (*standard*, *unfold split*, *rule P-simp-pp*, *rule pat-inf-var-conflict*)
from *True* **have** *ivc* ≠ [] **by** *auto*
then obtain *lx rx b ivc'* **where** *ivc*: *ivc* = (*lx*,*rx*,*b*) # *ivc'* **by** (*cases ivc*, *auto*)
hence (*lx*,*rx*,*b*) ∈ *set ivc* **by** *auto*
from *ivc-b*[*OF this*] **have** *mp-rx* (*rx*,*b*) ≠ {#} **unfolding** *inf-var-conflict-def* **by** *auto*
thus *M* ≠ {#} **unfolding** *M-def ivc pat-lr-def* **by** *auto*
next
{
fix *xl xr b*
assume (*xl*, *xr*, *b*) ∈ *set ivc*
from *ivc-b*[*OF this*] **have** *inf-var-conflict* (*mp-mset* (*mp-rx* ((*xr*, *b*)))) **by** *simp*
hence *inf-var-conflict* (*mp-mset* (*mp-lr* (*xl*, *xr*, *b*)))
unfolding *mp-lr-def inf-var-conflict-def* **by** *force*
}
thus *Ball* (*pat-mset M*) *inf-var-conflict* **unfolding** *M-def pat-lr-def* **by** *auto*
next
show ∀ *x*∈*tvars-pat* (*pat-mset* (*pat-lr P'*)). ¬ *inf-sort* (*snd x*)
proof
fix *y*
assume *y* ∈ *tvars-pat* (*pat-mset* (*pat-lr P'*))
from *this*[*unfolded tvars-pat-def pat-lr-def*, *simplified*] **obtain** *mp*
where *mp*: *mp* ∈ *set P'* **and** *y*: *y* ∈ *tvars-match* (*mp-mset* (*mp-lr mp*))
by *auto*
from *wf*[*unfolded wf-pat-lr-def*] *P' mp* **have** *wf*: *wf-lr3 mp* **by** *auto*
from *mp*[*unfolded P'-def*] **have** *mp*: *mp* ∈ *set no-ivc* **and** *fmp*: *?f mp* **by** *auto*
from *no-ivc-b*[*OF mp*] *True mp*
obtain *rx* **where** *mp-id*: *mp* = ([],*rx*,*False*)
and *ninf*: ¬ *inf-var-conflict* (*mp-mset* (*mp-rx* (*rx*, *False*)))
by (*cases mp*, *auto*)
note *fmp* = *fmp*[*unfolded mp-id snd-conv fst-conv*]
have *id*: *mp-lr mp* = *mp-rx* (*rx*,*False*) **unfolding** *mp-id mp-lr-def* **by** *auto*
from *y*[*unfolded id mp-rx-def List.maps-def tvars-match-def*]
obtain *x ts t* **where** *xts*: (*x*,*ts*) ∈ *set rx* **and** *t*: *t* ∈ *set ts* **and** *y*: *y* ∈

vars t **by** *force*
from *wf*[*unfolded mp-id wf-lr3-def split*]
have *wf-rx3* (*rx*, *False*) **by** *auto*
from *this*[*unfolded wf-rx3-def*] *xts True*
have *wf-ts3 ts* **and** *wf2*: *wf-rx2* (*rx*, *False*) **by** *auto*
from *this*[*unfolded wf-ts3-def*] **obtain** *z* **where** *z*: *Var z* ∈ *set ts* **by** *auto*
have *sort*: $\mathcal{T}(C,\mathcal{V})$ (*Var z*) = *Some* (*snd z*) **by** *simp*
from *fmp*[*rule-format*, *OF xts*]
have *is-singleton-list* (*map* $\mathcal{T}(C,\mathcal{V})$ *ts*) **by** *auto*
from *this*[*unfolded is-singleton-list singleton-def*] **obtain** *so*
where *set* (*map* $\mathcal{T}(C,\mathcal{V})$ *ts*) = {*so*} **by** *auto*
with *z sort*
have *single*: *set* (*map* $\mathcal{T}(C,\mathcal{V})$ *ts*) = {*Some* (*snd z*)} **by** *force*
from *wf2*[*unfolded wf-rx2-def fst-conv*] *xts*
have *wf2*: *wf-ts2 ts* **by** *auto*
from *this*[*unfolded wf-ts2-def*] *z* **obtain** *s* **where** *s*: *s* ∈ *set ts* **and** *sz*: *s* ≠ *Var z*
by (*cases ts*; *cases tl ts*, *auto*)
from *wf2*[*unfolded wf-ts2-def wf-ts-no-conflict-alt-def*]
have *no-conf*: *s* ∈ *set ts* ⟹ *t* ∈ *set ts* ⟹ *conflicts s t* ≠ *None* **for** *s t* **by** *auto*
from *s z xts* **have**
mem: (*Var z*, *Var x*) ∈ *mp-mset* (*mp-rx* (*rx*, *False*))
(*s*, *Var x*) ∈ *mp-mset* (*mp-rx* (*rx*, *False*))
unfolding *mp-rx-def List.maps-def* **by** *auto*
from *no-conf*[*OF z s*]
have *Conflict-Var* (*Var z*) *s z* **using** *sz* **by** (*cases s*, *auto simp*: *conflicts.simps*)
with *ninf mem* **have** *ninf* :¬ *inf-sort* (*snd z*)
unfolding *inf-var-conflict-def* **by** *blast*
define *σ* **where** *σ* = *snd z*
from *single t*
have *t*: *t* : *σ in* $\mathcal{T}(C,\mathcal{V})$ **unfolding** *hastype-def σ-def* **by** *auto*
from *t y ninf*[*folded σ-def*]
show ¬ *inf-sort* (*snd y*)
by (*rule finite-sort-imp-finite-sort-vars*)
qed
qed (*insert True*, *auto*)
have {# *pat-lr P′* #} + *P* = *add-mset* (*pat-lr P′*) *P* **by** *simp*
also have *to-list*: *pat-lr P′* = *pat-mset-list* (*map mp-lr-list P′*) **by** (*simp add*: *pat-mset-list-lr*)
finally have *steps*: (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*pat-mset-list* (*map mp-lr-list P′*)) *P*) ∈ ⇛$^+$
using *steps step* **by** (*simp add*: *pat-mset-list-lr*)
show *?thesis*
proof (*intro conjI impI*)
assume *res* = *New-Problems* (*n′*, *nl′*, *ps*)
from *res*[*unfolded this*]
have *id*: *n′* = *n nl′* = *nl″ ps* = [*map mp-lr-list P′*]

```
          by (auto simp: P'-def)
        show (add-mset (pat-mset-list p) P, mset (map pat-mset-list ps) + P) ∈
⇛⁺
          unfolding id using steps by auto
        show n ≤ n' unfolding id by auto
        have tvars-pat (⋃ (pat-list ‘ set ps)) ⊆ tvars-pat (pat-list (map mp-lr-list
P'))
          unfolding id by auto
      also have id2: pat-list (map mp-lr-list P') = pat-mset (pat-lr P') unfolding
to-list
          by (metis pat-mset-list)
        also have tvars-pat ... ⊆ tvars-pat (pat-mset (pat-lr p')) using P'
          unfolding tvars-pat-def pat-lr-def by force
        also have ... ⊆ {..<n} × S by fact
        finally show tvars-pat (⋃ (pat-list ‘ set ps)) ⊆ {..<n'} × S
          unfolding id .
        show Multiset.Ball (mset (map pat-mset-list ps) + P) (lvar-cond-pp nl')
        proof
          fix mps
          assume mps ∈# mset (map pat-mset-list ps) + P
          from this[unfolded id]
         have disj: mps = pat-mset-list (map mp-lr-list P') ∨ mps ∈# P by auto
          thus lvar-cond-pp nl' mps
          proof
            assume mps ∈# P
            with lvarsAll have lvar-cond-pp nl mps by auto
            with lvar-cond-mono[OF nl] show lvar-cond-pp nl' mps
              unfolding lvar-cond-pp-def id by auto
          next
            assume mps = pat-mset-list (map mp-lr-list P')
            also have ... = pat-lr P' by (rule pat-mset-list-lr)
             also have ... ⊆# pat-lr no-ivc unfolding P'-def pat-lr-def mset-map
mset-filter
              by (rule image-mset-subseteq-mono, rule multiset-filter-subset)
            also have ... ⊆# pat-lr p' unfolding p'-split by auto
            finally have lvars-pp mps ⊆ lvars-pp (pat-lr p')
              unfolding lvars-pp-def using mset-subset-eqD by fastforce
           with lvar-p' show ?thesis unfolding id lvar-cond-pp-def lvar-cond-def
by auto
          qed
        qed
      qed (insert res, auto)
    next
      case False
      hence ?ivc'-cond = False by auto
      note res = res[unfolded this if-False]
      show ?thesis
      proof (cases find-var improved no-ivc)
        case (Some x)
```

define *ps* **where** *ps* = *map* (λτ. *subst-pat-problem-list* τ *p'l*) (τ*s-list n x*)
have *id*: *pat-lr p'* = *pat-mset-list p'l* **unfolding** *p'l-def* **by** (*simp add*: *pat-mset-list-lr*)
have *subst*: *map* (λτ. *subst-pat-problem-mset* τ (*pat-lr p'*)) (τ*s-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*)
note *res* = *res*[*unfolded Let-def Some option.simps*, *folded p'l-def*]
from *res* **have** *res*: *res* = *New-Problems* (*n* + *m*, *nl''*, *ps*) **using** *ps-def* **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* = *Some*[*unfolded find-var-def*]
let *?concat* = *List.maps* (λ (*lx*,-). *lx*) *no-ivc*
have *disj*: *tvars-disj-pp* {*n*..<*n* + *m*} (*pat-mset* (*pat-lr p'*))
using *varsp'* **unfolding** *tvars-pat-def tvars-disj-pp-def tvars-match-def* **by** *force*
show *pat-lr p'* ⇒$_m$ *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* ∈ *set p'* **and** (*x*,*t*) ∈ *set* ((λ (*lx*,-). *lx*) *mp*) **using** *sub*
by (*auto simp*: *List.maps-def*)
then obtain *lx rx* **where** *mem*: (*lx*,*rx*) ∈ *set p'* **and** *xt*: (*x*,*t*) ∈ *set lx* **by** *auto*
from *wf mem* **have** *wf*: *wf-lx lx* **unfolding** *wf-pat-lr-def wf-lr3-def* **by** *auto*
with *xt* **have** *t*: *is-Fun t* **unfolding** *wf-lx-def* **by** *auto*
from *mem* **obtain** *p''* **where** *pat*: *pat-lr p'* = *add-mset* (*mp-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*) ∈# *mp-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*
define *flat-mps* **where** *flat-mps* = *List.maps* (*fst* ∘ *snd*) *no-ivc*
note *x* = *x*[*unfolded Nil list.simps Let-def*, *folded flat-mps-def*]
from *wf*[*unfolded wf-pat-lr-def*]

```
          show ?thesis
          proof (cases improved)
            case False
             from no-ivc obtain mp p'' where fp: no-ivc = mp # p'' by (cases
no-ivc) auto
             obtain lx rx b where mp: mp = (lx,rx,b) by (cases mp) auto
             from fp have hd: hd no-ivc = mp by auto
             from no-ivc-b[of mp, unfolded fp] mp
             have mp: mp = (lx,rx,False) by auto
             have mpp: mp ∈ set p' using arg-cong[OF fp, of set] sub by auto
             from mp Nil fp have lx = [] by (auto simp: List.maps-def)
             with mp have mp: mp = ([],rx,False) by auto
             note x = x[unfolded hd mp Let-def split]
            from wf mpp have wf: wf-lr3 mp and ne: ¬ empty-lr mp  unfolding
wf-pat-lr-def by auto
             from wf[unfolded wf-lr3-def mp split] mp
             have wf: wf-rx2 (rx, False) by (auto simp: wf-rx3-def)
             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: ¬ 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 ≠ t and conf: conflicts s t ≠ 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 ≠ [] by auto
             with x[unfolded rx list.simps list.sel split ts conf option.sel] False
             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) ∈# mp-rx (rx, False) (t, Var y) ∈#
mp-rx (rx,False)
               by (auto simp: mp-rx-def List.maps-def)
            with confl ninf have ¬ inf-sort (snd x) unfolding inf-var-conflict-def
by blast
             with sty confl rx have main: (s, Var y) ∈# mp-lr mp ∧ (t, Var y)
∈# mp-lr mp ∧ Conflict-Var s t x ∧ ¬ inf-sort (snd x)
          ∧ (improved ⟶ b) for b using False
               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 .
            next
             case impr: True
```

hence *improved = True* **by** *auto*
note *x = x*[*unfolded this if-True*]
let *?find = find* (λ*rx*. ∃ *t*∈*set* (*snd rx*). *is-Fun t*) *flat-mps*
from *x* **obtain** *rx* **where** *find*: *?find = Some rx* **by** (*cases ?find*; *force*)
from *this*[*unfolded find-Some-iff*]
obtain *t* **where** *rx*: *rx* ∈ *set flat-mps* **and** *t*: *t* ∈ *set* (*snd rx*) *is-Fun t* **by** *auto*
obtain *y ts* **where** *rx-id*: *rx* = (*y*,*ts*) **by** *force*
note *x = x*[*unfolded find option.simps rx-id split*]
from *rx*[*unfolded flat-mps-def List.maps-def*]
obtain *mp* **where** *mp-mem*: *mp* ∈ *set no-ivc* **and** *rx-mp*: *rx* ∈ *set* (*fst* (*snd mp*)) **by** *auto*
from *mp-mem sub* **have** *mp-mem-p'*: *mp* ∈ *set p'* **by** *auto*
then 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*)
obtain *lx rxs b* **where** *mp*: *mp* = (*lx*,*rxs*,*b*) **by** (*cases mp*, *auto*)
with *rx-mp* **have** *rx-rxs*: *rx* ∈ *set rxs* **by** *auto*
from *split-list*[*OF mp-mem*] *Nil mp*
have *lx*: *lx* = [] **unfolding** *List.maps-def* **by** *auto*
from *no-ivc-b*[*OF mp-mem*] *mp lx*
have *mp*: *mp* = ([],*rxs*,*False*) **and**
No-ivc: ¬ *inf-var-conflict* (*mp-mset* (*mp-rx* (*rxs*,*b*))) **by** *auto*
from *wf*[*unfolded wf-pat-lr-def*] *mp-mem sub*
have *wf-lr3 mp* **by** *auto*
from *this*[*unfolded wf-lr3-def mp split*]
have *wf-rx3* (*rxs*, *False*) **by** *auto*
from *this*[*unfolded wf-rx3-def fst-conv snd-conv*]
have *wf-rx2*: *wf-rx2* (*rxs*, *False*) **and** *Ball* (*snd* ‘ *set rxs*) *wf-ts3* **using** *impr* **by** *auto*
with *rx-mp*[*unfolded mp*] *rx-id*
have *wf-ts*: *wf-ts3 ts* **by** *auto*
from *this*[*unfolded wf-ts3-def*] **have** ∃ *u*. *u* ∈ *set ts* ∧ *is-Var u* ∧ *find is-Var ts* = *Some u*
by (*induct ts*, *auto*)
with *x* **have** *x*: *Var x* ∈ *set ts* **by** *auto*
from *t*[*unfolded rx-id*]
have *t*: *t* ∈ *set ts is-Fun t* **by** *auto*
show *?thesis*
proof (*rule pat-instantiate*[*of n mp-lr mp p''*,
OF - disjI2, *of Var x y t x*, *folded pat*, *OF disj*, *unfolded subst*],
intro conjI impI refl t)
show *cvar1*: *x* ∈ *set* (*the* (*conflicts* (*Var x*) *t*)) **using** *t* **by** (*cases t*, *auto simp*: *conflicts.simps*)
from *x rx-rxs rx-id* **have** *xy*: (*Var x*, *Var y*) ∈# *mp-rx* (*rxs*, *b*) **unfolding** *mp-rx-def List.maps-def* **by** *auto*
thus (*Var x*, *Var y*) ∈# *mp-lr mp*
unfolding *mp-lr-def mp split mp-rx-def* **by** *auto*

```
            from t rx-rxs rx-id have ty: (t, Var y) ∈# mp-rx (rxs, b) unfolding
mp-rx-def List.maps-def by auto
              thus (t, Var y) ∈# mp-lr mp
                unfolding mp-lr-def mp split mp-rx-def by auto
            from rx-rxs rx-id wf-rx2[unfolded wf-rx2-def] have wf-ts2 ts by auto
              from this[unfolded wf-ts2-def wf-ts-no-conflict-alt-def] x t
              show cvar2: conflicts (Var x) t ≠ None by auto
               from No-ivc[unfolded inf-var-conflict-def] xy ty cvar1 cvar2 show
¬ inf-sort (snd x) by blast
            qed
          qed
        qed
      qed
      have tvars: tvars-pat (⋃ (pat-list ‘ set ps)) ⊆ {..<n + m} × S
      proof (safe del: conjI)
        fix yn ι
        assume (yn,ι) ∈ tvars-pat (⋃ (pat-list ‘ set ps))
        then obtain pi mp
          where pi: pi ∈ set ps
            and mp: mp ∈ set pi and y: (yn,ι) ∈ tvars-match (set mp)
          unfolding tvars-pat-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: τ ∈ set (τs-list n x) and pi: pi = map (map
(subst-left τ)) p'l by auto
        from tau[unfolded τs-list-def]
        obtain info where infoCl: info ∈ set (Cl (snd x)) and tau: τ = τc n x
info by auto
        from Cl-len[of snd x] this(1) have len: length (snd info) ≤ m by force
       from mp[unfolded pi set-map] obtain mp' where mp': mp' ∈ set p'l and
mp: mp = map (subst-left τ) mp' by auto
        from y[unfolded mp tvars-match-def image-comp o-def set-map]
          obtain pair where ∗: pair ∈ set mp' (yn,ι) ∈ 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) ∈ set mp' and y: (yn,ι) ∈ vars (s ·
τ) unfolding subst-left-def by auto
        from y[unfolded vars-term-subst, simplified]
        obtain z where z: z ∈ vars s and y: (yn,ι) ∈ vars (τ z) by auto
        obtain f ss where info: info = (f,ss) by (cases info, auto)
        with len have len: length ss ≤ m by auto
       define ts :: ('f,-)term list where ts = map Var (zip [n..<n + length ss]
ss)
        from tau[unfolded τc-def info split]
        have tau: τ = subst x (Fun f ts) unfolding ts-def by auto
        from infoCl[unfolded Cl info]
        have f: f : ss → snd x in C by auto
        from C-sub-S[OF this] have ssS: set ss ⊆ S by simp
        from ssS
```

```
          have vars (Fun f ts) ⊆ {..< n + length ss} × S unfolding ts-def by
(auto simp: set-zip)
          also have ... ⊆ {..< n + m} × S using len by auto
          finally have subst: vars (Fun f ts) ⊆ {..< n + m} × S by auto
          show yn ∈ {..<n + m} ∧ ι ∈ S
          proof (cases z = x)
            case True
            with y subst tau show ?thesis by force
          next
            case False
            hence τ z = Var z unfolding tau by (auto simp: subst-def)
            with y have z = (yn,ι) by auto
            with z have y: (yn,ι) ∈ vars s by auto
           with st have (yn,ι) ∈ tvars-match (set mp') unfolding tvars-match-def
by force
           with mp' have (yn,ι) ∈ tvars-pat (set ' set p'l) unfolding tvars-pat-def
by auto
            also have ... = tvars-pat (pat-mset (pat-mset-list p'l))
                  by (rule arg-cong[of - - tvars-pat], auto simp: pat-mset-list-def
image-comp)
          also have ... = tvars-pat (pat-mset (pat-lr p')) unfolding id[symmetric]
by simp
            also have ... ⊆ {..<n} × S using varsp' .
            finally show ?thesis by auto
          qed
        qed
        {
          fix pp
          assume pp: pp ∈# mset (map pat-mset-list ps) + P
          have lvar-cond-pp nl'' pp
          proof (cases pp ∈# P)
            case True
            with lvarsAll have lvar-cond-pp nl pp by auto
            with lvar-cond-mono[OF nl] show ?thesis unfolding lvar-cond-pp-def
by auto
          next
            case False
             then obtain pp' where pp': pp' ∈ set ps and pp: pp = pat-mset-list
pp'
              using pp by auto
        from pp'[unfolded ps-def] obtain τ where pp': pp' = subst-pat-problem-list
τ p'l by auto
            have lvars-pp pp = lvars-pp (pat-lr p')
              unfolding pp pp' id
              unfolding lvars-pp-def lvars-mp-def
             by (force simp: subst-pat-problem-list-def subst-match-problem-list-def
subst-left-def pat-mset-list-def)
            thus ?thesis using lvar-p' unfolding lvar-cond-pp-def by auto
          qed
```

```
        }
        with tvars step steps res nl show ?thesis by auto
      next
        case None
          hence impr: improved and Nil: List.maps (λ(lx, uu). lx) no-ivc = []
unfolding find-var-def Let-def
          by (auto split: option.splits if-splits list.splits)
        from impr False have ivc = [] ∨ (∃ mp ∈ set no-ivc. fst mp ≠ []) by auto
        with Nil have ivc: ivc = [] unfolding List.maps-def by force
        {
          fix mp
          assume mp-mem: mp ∈ set no-ivc
         from no-ivc-b[OF mp-mem] obtain lx rx where mp: mp = (lx,rx,False)
by (cases mp, auto)
          from None[unfolded find-var-def] have no-ivc-lx: List.maps (λ(lx, uu).
lx) no-ivc = [] by (auto split: list.splits)
          from split-list[OF mp-mem] mp no-ivc-lx have lx: lx = [] by (cases lx,
auto simp: List.maps-def)
          with mp have mp: mp = ([],rx, False) by auto
          from impr have improved = True by simp
          note None = None[unfolded find-var-def no-ivc-lx list.simps this if-True
Let-def]
          from None mp-mem
          have ∀ a b. ((a, b) ∉ set (fst (snd mp))) ∨ (∀ t∈set b. is-Var t)
            by (auto simp: find-None-iff List.maps-def)
          from this[unfolded mp] have only-vars: (x, ts) ∈ set rx ⟹ t ∈ set ts
⟹ is-Var t for x ts t by auto
          from wf[unfolded wf-pat-lr-def] mp-mem have wf-lr3 mp using sub by
auto
          from this[unfolded wf-lr3-def mp split] have wf-rx3 (rx, False) by auto
          from this[unfolded wf-rx3-def] have wf-rx2 (rx, False) by auto
          from this[unfolded wf-rx2-def snd-conv fst-conv]
          have wf-ts: Ball (snd ` set rx) wf-ts2
            and no-inf: ¬ inf-var-conflict (mp-mset (mp-rx (rx, False)))
            and dist: distinct (map fst rx) by auto
          {
            fix x ts t
            assume xts: (x,ts) ∈ set rx and t: t ∈ set ts
            from only-vars[OF this] obtain y where ty: t = Var y by auto
            from xts wf-ts have wf-ts2 ts by auto
            from this[unfolded wf-ts2-def wf-ts-no-conflict-alt-def] t
             have len: 2 ≤ length ts and dist: distinct ts  and no-conf: ⋀ s. s ∈
set ts ⟹ conflicts s t ≠ None by auto
            from t len dist obtain s where s-ts: s ∈ set ts and s ≠ t by (cases
ts; cases tl ts; auto)
            from only-vars[OF xts this(1)] this(2) ty obtain z where s: s = Var
z and yz: y ≠ z by (cases s, auto)
          from t have trx: (t, Var x) ∈# mp-rx (rx, False) using xts unfolding
mp-rx-def List.maps-def by auto
```

from *s-ts* **have** *srx*: $(s,\ Var\ x) \in\#\ mp\text{-}rx\ (rx,\ False)$ **using** *xts* **unfolding** *mp-rx-def List.maps-def* **by** *auto*
have *conflicts s t* = (*if snd z* = *snd y then Some* [*z*, *y*] *else None*) **unfolding** *s ty conflicts.simps* **using** *yz* **by** *auto*
with *no-conf*[*OF s-ts*] **have** *conflicts s t* = *Some* [*z*,*y*] **by** (*auto split*: *if-splits*)
with *no-inf*[*unfolded inf-var-conflict-def*, *simplified*, *rule-format*, *OF trx srx*] *yz*
have ¬ *inf-sort* (*snd y*) **by** (*cases y*, *auto*)
with *ty* **have** ∃ *y*. *t* = *Var y* ∧ ¬ *inf-sort* (*snd y*) **by** *auto*
} **note** *only-fin-sort-vars* = *this*
with *mp dist wf-ts* ‹*wf-rx3* (*rx*, *False*)› **have** ∃ *rx*. *mp* = ([], *rx*, *False*) ∧ *Ball* (*snd* ‘ *set rx*) *wf-ts2* ∧ *distinct* (*map fst rx*) ∧
wf-rx3 (*rx*, *False*) ∧
(∀ *x ts t*. (*x*,*ts*) ∈ *set rx* ⟶ *t* ∈ *set ts* ⟶ (∃ *y*. *t* = *Var y* ∧ ¬ *inf-sort* (*snd y*)))
by *blast*
} **note** *no-ivc-probs* = *this*

have *split-p'*: *pat-lr p'* = *pat-lr ivc* + *pat-lr no-ivc* **unfolding** *Notf f*
unfolding *pat-lr-def* **by** (*induct p'*, *auto*)
hence *p'-no-ivc*: *pat-lr p'* = *pat-lr no-ivc* **unfolding** *ivc pat-lr-def* **by** *auto*

let *?fvf* = *map* (*map* (*map-prod id* (*map the-Var*)) ∘ *fst* ∘ *snd*) *no-ivc*
let *?pat* = λ *p*. *pat-mset-list* (*pat-of-var-form-list p*)
note *res* = *res*[*unfolded None option.simps Let-def*]
from *steps p'-no-ivc*
have (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*pat-lr no-ivc*) *P*) ∈ ⇛* **by** *auto*
also have *equiv*: *pat-lr no-ivc* = *?pat ?fvf*
unfolding *pat-lr-def pat-of-var-form-list-def match-of-var-form-list-def pat-mset-list-def map-map o-def mp-lr-def*
proof (*intro arg-cong*[*of - - mset*] *map-cong refl*)
fix *mp*
assume *mp* ∈ *set no-ivc*
from *no-ivc-probs*[*OF this*] **obtain** *rx* **where** *mp*: *mp* = ([], *rx*, *False*)
and *rx*: ⋀ *x tx t*. (*x*,*tx*) ∈ *set rx* ⟹ *t* ∈ *set tx* ⟹ ∃ *y*. *t* = *Var y* **by** *metis*
have *triv*: *mp-list* (*mp-lx* ([] :: ((*nat* × *'s*) × (*'f*, *'v*) *Term.term*) *list*)) + *M* = *M* **for** *M* **by** *auto*
show (*case mp of* (*lx*, *rx*) ⇒ *mp-list* (*mp-lx lx*) + *mp-rx rx*) =
mp-list
(*concat*
(*map* (λ*x*. *case map-prod id* (*map the-Var*) *x of* (*x*, *xa*) ⇒ *map* (λ*v*. (*Var v*, *Var x*)) *xa*)
(*fst* (*snd mp*))))
unfolding *mp split fst-conv snd-conv mp-rx-def List.maps-def triv*
by (*rule arg-cong*[*of - - λ xs*. *mset* (*concat xs*)], *intro map-cong refl*, *insert rx*, *force*)

qed
finally have *steps'*: (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*?pat ?fvf*) *P*) ∈ ⇛* .
have *fvf-res*: *finite-var-form-pat C* (*pat-mset* (*?pat ?fvf*))
unfolding *finite-var-form-pat-def equiv*[*symmetric*]
proof
fix *Mp*
assume *Mp* ∈ *pat-mset* (*pat-lr no-ivc*)
from *this*[*unfolded pat-lr-def*]
obtain *mp* **where** *mp-mem*: *mp* ∈ *set no-ivc* **and** *Mp*: *Mp* = *mp-mset* (*mp-lr mp*) **by** *auto*
from *no-ivc-probs*[*OF mp-mem*] **obtain** *rx* **where**
mp: *mp* = ([], *rx*, *False*) **and**
dist: *distinct* (*map fst rx*) **and**
wf-ts: *Ball* (*snd* ‘ *set rx*) *wf-ts2* **and**
no-inf: ⋀ *x ts t*. (*x*, *ts*) ∈ *set rx* ⟹ *t* ∈ *set ts* ⟹ (∃ *y*. *t* = *Var y* ∧ ¬ *inf-sort* (*snd y*)) **by** *auto*
show *finite-var-form-match C Mp*
unfolding *finite-var-form-match-def var-form-match-def*
proof (*intro conjI allI impI subsetI*)
fix *l x*
assume *xl*: (*Var x*, *l*) ∈ *Mp*
with *Mp mp mp-mem sub* **have** *x* ∈ *tvars-pat* (*pat-mset* (*pat-lr p'*))
apply (*auto simp*: *tvars-pat-def pat-lr-def mp-lr-def mp-rx-def tvars-match-def intro*!: *bexI*[*of - mp*])
by (*metis case-prod-conv term.set-intros*(*3*))
with *varsp'* **have** *sxS*: *snd x* ∈ *S* **by** *auto*
from *xl*[*unfolded Mp mp split mp-lr-def mp-rx-def List.maps-def*]
obtain *ts y* **where** *yts*: (*y*,*ts*) ∈ *set rx* **and** *xts*: *Var x* ∈ *set ts* **and** *l*: *l* = *Var y* **by** *auto*
from *no-inf*[*OF yts xts*] **have** ¬ *inf-sort* (*snd x*) **by** *auto*
then show *finite-sort C* (*snd x*) **by** (*simp add*: *inf-sort*[*OF sxS*])
fix *z*
assume (*Var z*, *l*) ∈ *Mp*
from *this*[*unfolded Mp mp split mp-lr-def mp-rx-def List.maps-def*] *l*
obtain *ts'* **where** *yts'*: (*y*,*ts'*) ∈ *set rx* **and** *zts*: *Var z* ∈ *set ts'* **by** *auto*
from *dist yts yts'* **have** *ts'* = *ts* **by** (*metis eq-key-imp-eq-value*)
with *zts* **have** *zts*: *Var z* ∈ *set ts* (**is** *?z* ∈ -) **by** *auto*
from *wf-ts yts* **have** *wf-ts2 ts* **by** *auto*
from *this*[*unfolded wf-ts2-def wf-ts-no-conflict-alt-def*] *xts zts*
have *conflicts* (*Var x*) *?z* ≠ *None* **by** *blast*
thus *snd x* = *snd z* **unfolding** *conflicts.simps* **by** (*auto split*: *if-splits*)
next
fix *pair*
assume *pair* ∈ *Mp*
from *this*[*unfolded Mp mp-lr-def mp split*]
have *pair* ∈# *mp-rx* (*rx*, *False*) **by** *auto*
from *this*[*unfolded mp-rx-def List.maps-def*]
obtain *x ts t* **where** (*x*,*ts*) ∈ *set rx t* ∈ *set ts* **and** *pair*: *pair* = (*t*, *Var*

```
x) by auto
            with no-inf[OF this(1−2)]
            show pair ∈ range (map-prod Var Var) by auto
          qed
        qed
        show ?thesis using res
        proof (intro conjI, force, force, intro impI conjI)
          assume res = Fin-Var-Form fvf
          with res have id: fvf = ?fvf by simp
          show improved by fact
         show (add-mset (pat-mset-list p) P, add-mset (?pat fvf) P) ∈ ⇛* using
steps′ id by auto
           show finite-var-form-pat C (pat-list (pat-of-var-form-list fvf)) using
fvf-res
           unfolding id pat-mset-list by auto
        show Ball (set fvf) (distinct ∘ map fst) unfolding id using no-ivc-probs
by force
         show Ball (set (concat fvf)) (distinct ∘ snd)
         proof
           fix xvs
           assume xvs ∈ set (concat fvf)
           from this[unfolded id] obtain c where
             c: c ∈ set no-ivc and xvs: xvs ∈ map-prod id (map the-Var) ‘ set (fst
(snd c)) by auto
           from no-ivc-probs[OF c] obtain rx where ∗:
             c = ([], rx, False)
             Ball (snd ‘ set rx) wf-ts2
               (∀ x ts t. (x, ts) ∈ set rx ⟶ t ∈ set ts ⟶ (∃ y. t = Var y ∧ ¬
inf-sort (snd y))) by blast
           from xvs[unfolded ∗(1)]
           have xvs: xvs ∈ map-prod id (map the-Var) ‘ set rx by auto
           then obtain x ts where mem: (x,ts) ∈ set rx and xvs: xvs = (x,map
the-Var ts) by auto
           from ∗(2) mem have wf-ts2 ts by auto
           hence dist: distinct ts unfolding wf-ts2-def by auto
           show (distinct ∘ snd) xvs unfolding xvs o-def snd-conv
               distinct-map
           proof (rule conjI[OF dist])
             show inj-on the-Var (set ts)
               by (auto simp: inj-on-def dest!: ∗(3)[rule-format, OF mem])
           qed
         qed
        qed
      qed
    qed
  qed
qed
```

lemma *non-uniq-image-diff*: ¬ *UNIQ* (α ‘ *set vs*) ⟷ (∃ *v* ∈ *set vs*. ∃ *w* ∈ *set vs*. α *v* ≠ α *w*)
by (*smt* (*verit*, *ccfv-SIG*) *Uniq-def image-iff*)

lemma *pat-complete-via-idl-solver*:
assumes *impr*: *improved*
and *fvf*: *finite-var-form-pat C* (*pat-list pp*)
and *wf*: *wf-pat* (*pat-list pp*)
and *pp*: *pp* = *pat-of-var-form-list fvf*
and *dist*: *Ball* (*set fvf*) (*distinct o map fst*)
and *dist2*: *Ball* (*set* (*concat fvf*)) (*distinct o snd*)
and *cnf*: *cnf* = *map* (*map snd*) *fvf*
shows *pat-complete C* (*pat-list pp*) ⟷ ¬ *fidl-solver* (*bounds-list* (*cd-sort* ∘ *snd*) *cnf*, *dist-pairs-list cnf*)
proof−
let *?S* = *S*
note *vf* = *finite-var-form-imp-of-var-form-pat*[*OF fvf*]
have *var-conv*: *set* (*concat* (*concat cnf*)) = *tvars-pat* (*pat-list pp*)
unfolding *cnf pp*
by (*force simp*: *tvars-pat-def pat-list-def tvars-match-def pat-of-var-form-list-def match-of-var-form-list-def*)
from *wf*[*unfolded wf-pat-iff*] *cd*
have *cd-conv*: *v* ∈ *tvars-pat* (*pat-list pp*) ⟹ *cd-sort* (*snd v*) = *card-of-sort C* (*snd v*)
for *v* **by** *auto*
define *cd* :: *nat* × *'s* ⇒ *nat* **where** *cd* = (*cd-sort* ∘ *snd*)
define *S* **where** *S* = *set* (*concat* (*concat cnf*))
{
fix *v vs c*
assume *c* ∈ *set cnf vs* ∈ *set c v* ∈ *set vs*
hence *v* ∈ *S* **unfolding** *S-def* **by** *auto*
} note *in-S* = *this*
have *pat-complete C* (*pat-list pp*) ⟷
(∀ α. (∀ *v*∈ *S*. α *v* < *cd v*) ⟶ (∃ *c*∈*set cnf*. ∀ *vs*∈*set c*. *UNIQ* (α ‘ *set vs*)))
by (*unfold S-def pat-complete-via-cnf*[*OF fvf pp dist cnf*] *var-conv*, *simp add*: *cd-conv cd-def*)
also have . . . ⟷ ¬ (∃ α. (∀ *v*∈ *S*. α *v* < *cd v*) ∧ (∀ *c* ∈ *set cnf*. ∃ *vs*∈*set c*. ¬ *UNIQ* (α ‘ *set vs*))) (**is** - ⟷ ¬ *?f*) **by** *blast*
also have *?f* ⟷ (∃ α. (∀ *v*∈ *S*. α *v* < *cd v*) ∧ (∀ *c*∈*set cnf*. ∃ *vs*∈*set c*. ∃ *v*∈*set vs*. ∃ *w*∈*set vs*. α *v* ≠ α *w*)) (**is** - ⟷ (∃ α. *?fN* α))
unfolding *non-uniq-image-diff* **..**
also have . . . ⟷ (∃ α. (∀ *v*∈ *S*. *0* ≤ α *v* ∧ α *v* < *int* (*cd v*)) ∧ (∀ *c*∈*set cnf*. ∃ *vs*∈*set c*. ∃ *v*∈*set vs*. ∃ *w*∈*set vs*. α *v* ≠ α *w*)) (**is** - ⟷ (∃ α. *?fZ* α))
proof
assume ∃ α. *?fN* α
then obtain α **where** *?fN* α **by** *blast*
hence *?fZ* (*int o* α) **unfolding** *o-def* **by** *auto*
thus ∃ α. *?fZ* α **by** *blast*
next

assume $\exists\ \alpha.\ ?fZ\ \alpha$
then obtain α **where** *alpha*: $?fZ\ \alpha$ **by** *blast*
have $?fN\ (nat\ o\ \alpha)$ **unfolding** *o-def*
proof (*intro conjI ballI*)
show $v \in S \Longrightarrow nat\ (\alpha\ v) < cd\ v$ **for** v **using** *alpha* **by** *auto*
fix c
assume *c*: $c \in set\ cnf$
with *alpha* **obtain** *vs v w* **where** *vs*: $vs \in set\ c$ **and** *v*: $v \in set\ vs$ **and** *w*: $w \in set\ vs$ **and** *diff*: $\alpha\ v \neq \alpha\ w$
by *auto*
from *in-S*[*OF c vs*] *v w* **have** $v \in S\ w \in S$ **by** *auto*
with *alpha* **have** $\alpha\ v \geq 0\ \alpha\ w \geq 0$ **by** *auto*
with *diff* **have** $nat\ (\alpha\ v) \neq nat\ (\alpha\ w)$ **by** *simp*
with *vs v w* **show** $\exists\, vs \in set\ c.\ \exists\, v \in set\ vs.\ \exists\, w \in set\ vs.\ nat\ (\alpha\ v) \neq nat\ (\alpha\ w)$ **by** *auto*
qed
thus $\exists\ \alpha.\ ?fN\ \alpha$ **by** *blast*
qed
also have $\ldots \longleftrightarrow (\exists\ \alpha.\ (\forall\, v \in S.\ 0 \leq \alpha\ v \wedge \alpha\ v \leq int\ (cd\ v) - 1) \wedge (\forall\, c \in set\ cnf.\ \exists\, vs \in set\ c.\ \exists\, v \in set\ vs.\ \exists\, w \in set\ vs.\ \alpha\ v \neq \alpha\ w))$
by *auto*
also have $\ldots = (\exists\, \alpha.\ (\forall\, (v,\ b) \in set\ (bounds\text{-}list\ cd\ cnf).\ 0 \leq \alpha\ v \wedge \alpha\ v \leq b) \wedge (\forall\, c \in set\ (dist\text{-}pairs\text{-}list\ cnf).\ \exists\, (v,\ w) \in set\ c.\ \alpha\ v \neq \alpha\ w))$
unfolding *bounds-list-def Let-def S-def*[*symmetric*] *set-map set-remdups*
proof (*intro arg-cong*[*of - - Ex*] *ext arg-cong2*[*of - - - - (∧)*], *force*)
fix $\alpha :: - \Rightarrow int$
show $(\forall\, c \in set\ cnf.\ \exists\, vs \in set\ c.\ \exists\, v \in set\ vs.\ \exists\, w \in set\ vs.\ \alpha\ v \neq \alpha\ w) = (\forall\, c \in set\ (dist\text{-}pairs\text{-}list\ cnf).\ \exists\, (v,\ w) \in set\ c.\ \alpha\ v \neq \alpha\ w)$
unfolding *diff-pairs-of-list dist-pairs-list-def List.maps-def set-map image-comp set-concat o-def*
by *force*
qed
also have $\ldots = fidl\text{-}solvable\ (bounds\text{-}list\ cd\ cnf,\ dist\text{-}pairs\text{-}list\ cnf)$
unfolding *fidl-solvable-def split* **..**
also have $\ldots = fidl\text{-}solver\ (bounds\text{-}list\ cd\ cnf, dist\text{-}pairs\text{-}list\ cnf)$
proof (*rule sym*, *rule fidl-solver*[*OF ‹improved›*, *unfolded finite-idl-solver-def*, *rule-format*])
show *fidl-input* (*bounds-list cd cnf*, *dist-pairs-list cnf*) **unfolding** *fidl-input-def split*
proof (*intro conjI allI impI*)
show $(x,\ y) \in set\ (concat\ (dist\text{-}pairs\text{-}list\ cnf)) \Longrightarrow z \in \{x,\ y\} \Longrightarrow z \in fst\ `\ set\ (bounds\text{-}list\ cd\ cnf)$ **for** *x y z*
unfolding *dist-pairs-list-def bounds-list-def List.maps-def set-concat set-map image-comp o-def*
set-pairs-of-list **by** *force*
show *distinct* (*map fst* (*bounds-list cd cnf*)) **unfolding** *bounds-list-def Let-def map-map o-def*
by *auto*
show $\bigwedge v\ w\ b1\ b2.$

```
    (v, b1) ∈ set (bounds-list cd cnf) ⟹
    (w, b2) ∈ set (bounds-list cd cnf) ⟹ snd v = snd w ⟹ b1 = b2
     unfolding bounds-list-def Let-def by (auto simp: cd-def)
   {
    fix v b
    assume (v, b) ∈ set (bounds-list cd cnf)
    from this[unfolded bounds-list-def]
    have v: v ∈ tvars-pat (pat-list pp) and b: b = int (cd v) − 1 by (auto simp
flip: var-conv)
     from cd-conv[OF v] b have b: b = int (card-of-sort C (snd v)) − 1 by
(auto simp: cd-def)
    from wf[unfolded wf-pat-iff, rule-format, OF v]
    have vS: snd v ∈ ?S by auto
    from not-empty-sort[OF this]
    have nE: ¬ empty-sort C (snd v) .
    from v[unfolded tvars-pat-def tvars-match-def]
    obtain mp t l where mp: mp ∈ pat-list pp and tl: (t,l) ∈ mp and vt: v ∈
vars t by auto
   from fvf[unfolded finite-var-form-pat-def] mp have mp: finite-var-form-match
C mp by auto
    note mp = mp[unfolded finite-var-form-match-def]
    from mp[unfolded var-form-match-def] tl obtain x where t: t = Var x by
auto
    with vt tl have vl: (Var v, l) ∈ mp by auto
    with mp have finite-sort C (snd v) by blast
      with nE have card-of-sort C (snd v) > 0 unfolding empty-sort-def
finite-sort-def card-of-sort-def
      by fastforce
    thus 0 ≤ b unfolding b by simp
   }
   fix v w
   assume (v, w) ∈ set (concat (dist-pairs-list cnf))
   from this[unfolded dist-pairs-list-def cnf List.maps-def, simplified]
   obtain c x vs where c: c ∈ set fvf and xvs: (x,vs) ∈ set c and vw: (v, w) ∈
set (pairs-of-list vs)
     by auto
   from dist2 c xvs have dist2: distinct vs by force
   from vw[unfolded set-pairs-of-list]
   obtain i where v: v = vs ! i and w: w =  vs ! Suc i and i: Suc i < length
vs by auto
   from dist2 v w i show v ≠ w unfolding distinct-conv-nth by simp

   from v w i have vw: v ∈ set vs w ∈ set vs by auto
  from fvf[unfolded pp finite-var-form-pat-def pat-list-def pat-of-var-form-list-def]
c
   have finite-var-form-match C (set (match-of-var-form-list c)) by auto
   from this[unfolded finite-var-form-match-def, THEN conjunct2, THEN con-
junct1, rule-format, of v Var x w]
   show snd v = snd w using vw xvs unfolding match-of-var-form-list-def by
```

auto
  qed
 qed
 finally show *?thesis* **unfolding** *cd-def* **.**
qed

The soundness property of the implementation, proven by induction on the relation that was also used to prove termination of ⇛. Note that we cannot perform induction on ⇛ here, since applying a decision procedure for finite-var-form problems does not correspond to a ⇛-step.

lemma *pats-impl*: **assumes** ∀ *p* ∈ *set ps*. *tvars-pat* (*pat-list p*) ⊆ {..<*n*} × *S*
 and *Ball* (*set ps*) (λ *pp*. *lvar-cond-pp nl* (*pat-mset-list pp*))
 and ∀ *pp* ∈ *pat-list* ‘ *set ps*. *wf-pat pp*
 shows *Pats-impl n nl ps* = *pats-complete C* (*pat-list* ‘ *set ps*)
proof (*insert assms*, *induct ps arbitrary*: *n nl rule*: *wf-induct*[*OF wf-inv-image*[*OF wf-trancl*[*OF wf-rel-pats*]], *of pats-mset-list*])
 case (*1 ps n nl*)
 note *IH* = *mp*[*OF spec*[*OF mp*[*OF spec*[*OF mp*[*OF spec*[*OF 1*(*1*)]]]]]]
 note *wf* = *1*(*4*)
 show *?case*
 proof (*cases ps*)
  case *Nil*
  show *?thesis* **unfolding** *pats-impl.simps*[*of - - - n nl 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 nl ps*, *unfolded Cons list.simps*, *folded Cons*]
  from *1*(*2*)[*rule-format*, *of p*] *Cons* **have** *tvars-pat* (*pat-list p*) ⊆ {..<*n*} × *S* **by** *auto*
  note *pat-impl* = *pat-impl*[*OF refl this*]
  from *1*(*3*) **have** ∀ *pp* ∈# *add-mset* (*pat-mset-list p*) (*pats-mset-list ps1*). *lvar-cond-pp nl pp*
  unfolding *Cons* **by** *auto*
  note *pat-impl* = *pat-impl*[*OF this*, *folded id*]
  let *?step* = (⇛) :: ((*'f*,*'v*,*'s*)*pats-problem-mset* × (*'f*,*'v*,*'s*)*pats-problem-mset*)*set*

  {
  from *rel-P-trans* **have** *single*: *?step* ⊆ (≺*mul*)^−*1*
  unfolding *P-step-def* **by** *auto*
  have (*s*,*t*) ∈ *?step*^+ ⟹ (*t*,*s*) ∈ (≺*mul*)^+ (*s*,*t*) ∈ *?step*^* ⟹ (*t*,*s*) ∈ (≺*mul*)^* **for** *s t*
  using *trancl-mono*[*OF - single*]
  apply (*metis converse-iff trancl-converse*)
  using *rtrancl-converse rtrancl-mono*[*OF single*]
  by *auto*
  } note *steps-to-rel* = *this*

from *wf* **have** *wf-pats* (*pat-list* ‘ *set ps*) **unfolding** *wf-pats-def* **by** *auto*
note *steps-to-equiv* = *P-steps-pcorrect*[*OF this*[*folded pats-mset-list*]]
show *?thesis*
proof (*cases Pat-impl n nl p*)
case *Incomplete*
with *res* **have** *res*: *Pats-impl n nl ps* = *False* **by** *auto*
from *pat-impl*(*1*)[*OF Incomplete*]
obtain *p′* **where** *steps*: (*pats-mset-list ps*, *add-mset p′* (*pats-mset-list ps1*)) ∈ ⇛* **and** *fail*: *pat-fail p′*
by *auto*
show *?thesis*
proof (*cases add-mset p′* (*pats-mset-list ps1*) = *bottom-mset*)
case *True*
with *res P-steps-pcorrect*[*OF - steps*, *unfolded pats-mset-list*] *wf*
show *?thesis* **by** (*auto simp*: *wf-pats-def*)
next
case *False*
from *P-failure*[*OF fail False*]
have (*add-mset p′* (*pats-mset-list ps1*), *bottom-mset*) ∈ ⇛ **unfolding** *P-step-def* **by** *auto*
with *steps* **have** (*pats-mset-list ps*, *bottom-mset*) ∈ ⇛* **by** *auto*
from *steps-to-equiv*[*OF this*] *res* **show** *?thesis* **unfolding** *pats-mset-list* **by** *simp*
qed
next
case (*New-Problems triple*)
then obtain *n2 nl2 ps2* **where** *Some*: *Pat-impl n nl p* = *New-Problems* (*n2*,*nl2*,*ps2*) **by** (*cases triple*) *auto*
with *res* **have** *res*: *Pats-impl n nl ps* = *Pats-impl n2 nl2* (*ps2* @ *ps1*) **by** *auto*
from *pat-impl*(*2*)[*OF Some*]
have *steps*: (*pats-mset-list ps*, *mset* (*map pat-mset-list* (*ps2* @ *ps1*))) ∈ ⇛+
and *vars*: *tvars-pat* (⋃ (*pat-list* ‘ *set ps2*)) ⊆ {..<*n2*} × *S*
and *lvars*: (∀ *pp*∈#*mset* (*map pat-mset-list ps2*) + *pats-mset-list ps1*. *lvar-cond-pp nl2 pp*)
and *n2*: *n* ≤ *n2* **by** *auto*
from *steps-to-rel*(*1*)[*OF steps*] **have** *rel*: (*ps2* @ *ps1*, *ps*) ∈ *inv-image* (≺*mul*+) *pats-mset-list*
by *auto*
have *vars*: ∀ *p*∈*set* (*ps2* @ *ps1*). *tvars-pat* (*pat-list p*) ⊆ {..<*n2*} × *S*
proof
fix *p*
assume *p* ∈ *set* (*ps2* @ *ps1*)
hence *p* ∈ *set ps2* ∨ *p* ∈ *set ps1* **by** *auto*
thus *tvars-pat* (*pat-list p*) ⊆ {..<*n2*} × *S*
proof
assume *p* ∈ *set ps2*
hence *tvars-pat* (*pat-list p*) ⊆ *tvars-pat* (⋃ (*pat-list* ‘ *set ps2*))
unfolding *tvars-pat-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*] *n2* **show** *?thesis* **by** *auto*
qed
qed
have *lvars*: $\forall$ *pp*$\in$*set* (*ps2* @ *ps1*). *lvar-cond-pp nl2* (*pat-mset-list pp*)
using *lvars* **unfolding** *lvar-cond-pp-def* **by** *auto*
note *steps-equiv* = *steps-to-equiv*[*OF trancl-into-rtrancl*[*OF steps*]]
from *steps-equiv* **have** *wf-pats* (*pats-mset* (*mset* (*map pat-mset-list* (*ps2* @ *ps1*)))) **by** *auto*
hence *wf2*: *Ball* (*pat-list* ‘ *set* (*ps2* @ *ps1*)) *wf-pat* **unfolding** *wf-pats-def pats-mset-list*[*symmetric*]
by *auto*
have *Pats-impl n nl ps* = *Pats-impl n2 nl2* (*ps2* @ *ps1*) **unfolding** *res* **by** *simp*
also have ... = *pats-complete C* (*pat-list* ‘ *set* (*ps2* @ *ps1*))
using *mp*[*OF IH*[*OF rel vars lvars*] *wf2*] **.**
also have ... = *pats-complete C* (*pat-list* ‘ *set ps*) **using** *steps-equiv*
unfolding *pats-mset-list*[*symmetric*] **by** *auto*
finally show *?thesis* **.**
next
case *FVF*: (*Fin-Var-Form fvf*)
let *?pat* = λ *p*. *pat-mset-list* (*pat-of-var-form-list p*)
let *?pat′* = λ *p*. *pat-list* (*pat-of-var-form-list p*)
from *pat-impl*(*3*)[*OF FVF*]
have *steps*: (*pats-mset-list ps*, *add-mset* (*?pat fvf*) (*pats-mset-list ps1*)) $\in \Rrightarrow^*$
and *ifvf*: *improved finite-var-form-pat C* (*?pat′ fvf*)
and *dist*: *Ball* (*set fvf*) (*distinct* $\circ$ *map fst*)
and *dist2*: *Ball* (*set* (*concat fvf*)) (*distinct* $\circ$ *snd*) **by** *auto*
have *wf-pats* (*pats-mset* (*pats-mset-list ps*))
using *wf* **unfolding** *wf-pats-def pats-mset-list* **.**
from *P-steps-pcorrect*[*OF this steps*]
have *wf′*: *wf-pats* (*pats-mset* (*add-mset* (*?pat fvf*) (*pats-mset-list ps1*)))
and *red*: *pats-complete C* (*pats-mset* (*pats-mset-list ps*)) =
pats-complete C (*pats-mset* (*add-mset* (*?pat fvf*) (*pats-mset-list ps1*)))
and *wf-pat* (*pat-mset* (*?pat fvf*)) **unfolding** *wf-pats-def* **by** *auto*
from *this*(*3*)[*unfolded pat-mset-list*]
have *wf-fvf*: *wf-pat* (*pat-list* (*pat-of-var-form-list fvf*)) **.**
have (*pats-mset-list ps1*, *add-mset* (*?pat fvf*) (*pats-mset-list ps1*)) $\in \prec mul$
unfolding *rel-pats-def* **by** (*simp add*: *subset-implies-mult*)
with *steps-to-rel*(*2*)[*OF steps*]
have (*pats-mset-list ps1*, *pats-mset-list ps*) $\in \prec mul^+$ **by** *auto*
hence (*ps1*, *ps*) $\in$ *inv-image* ($\prec mul^+$) *pats-mset-list* **by** *auto*
note *IH* = *IH*[*OF this*]
from *1(2) Cons* **have** $\forall$ *p*$\in$*set ps1*. *tvars-pat* (*pat-list p*) $\subseteq$ {..<*n*} $\times$ *S* **by** *auto*
note *IH* = *IH*[*OF this*]

```
      from 1(3) Cons have ∀pp∈set ps1. lvar-cond-pp nl (pat-mset-list pp) by
auto
      note IH = IH[OF this]
      with 1(4) Cons have IH: Pats-impl n nl ps1 = pats-complete C (pat-list ‘
set ps1) by auto
      note via-idl = pat-complete-via-idl-solver[OF ifvf wf-fvf refl dist dist2 refl]
      let ?cnf = (map (map snd) fvf)
      from FVF res have Pats-impl n nl ps =
          (¬ fidl-solver (bounds-list (cd-sort ∘ snd) ?cnf, dist-pairs-list ?cnf) ∧
Pats-impl n nl ps1)
        by (auto simp: Let-def)
    also have ... = (pat-complete C (pat-list (pat-of-var-form-list fvf)) ∧ pats-complete
C (pat-list ‘ set ps1))
        unfolding via-idl IH by simp
      also have ... = pats-complete C (pats-mset (pats-mset-list ps))
        unfolding steps-to-equiv[OF steps, THEN conjunct2]
         by (smt (z3) add-mset-commute comp-def image-iff insert-noteq-member
mset-add pat-mset-list
           pats-mset-list union-single-eq-member)
      also have ... = pats-complete C (pat-list ‘ set ps)
        unfolding pats-mset-list ..
      finally show ?thesis .
    qed
  qed
qed
```

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

```
corollary pat-complete-impl:
  assumes wf: snd ‘ ⋃ (vars ‘ fst ‘ set (concat (concat P))) ⊆ S
  shows Pat-complete-impl (P :: ('f,'v,'s)pats-problem-list) ⟷ pats-complete C
(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-match-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: ∀p∈set P. tvars-pat (pat-list p) ⊆ {..<n} × S
  proof (safe)
    fix p x ι
    assume p: p ∈ set P and xp: (x,ι) ∈ tvars-pat (pat-list p)
   hence x ∈ set ?l unfolding List.maps-def tvars-pat-def tvars-match-def pat-list-def

      by force
    from max-list[OF this] have x < n unfolding n-def by auto
    thus x < n by auto
    from xp p wf
    show ι ∈ S by (auto simp: wf-pat-iff)
```

```
  qed
  show ?thesis
  proof (cases improved)
    case False
    have 0: ∀ p∈set P. lvar-cond-pp 0 (pat-mset-list p)
      unfolding lvar-cond-pp-def lvar-cond-def allowed-vars-def using False by
auto
    have Pat-complete-impl P = Pats-impl n 0 P
      unfolding pat-complete-impl-def n-def Let-def using False by auto
    from pats-impl[OF n 0 wf, folded this]
    show ?thesis .
  next
    case True
    let ?r-mp = map (apsnd (map-vars renVar))
    let ?r = map ?r-mp
    let ?Q = map ?r P
    have Pat-complete-impl P = Pats-impl n 0 ?Q
      unfolding pat-complete-impl-def n-def Let-def using True by auto
    also have ... = pats-complete C (pat-list ` set ?Q)
    proof (rule pats-impl)
      show ∀ p ∈ set ?Q. tvars-pat (pat-list p) ⊆ {..<n} × S
      proof
        fix rp
        assume rp ∈ set ?Q
        then obtain p where p: p ∈ set P and rp: rp = ?r p by auto
        have id: tvars-pat (pat-list rp) = tvars-pat (pat-list p)
          unfolding pat-list-def rp tvars-pat-def tvars-match-def by force
        with n p show tvars-pat (pat-list rp) ⊆ {..<n} × S by auto
      qed
      show Ball (pat-list ` set ?Q) wf-pat
      proof −
        {
          fix rp
          assume rp: rp ∈ set ?Q
          then obtain p where p ∈ set P and rp: rp = ?r p by auto
          from this(1) wf have wf-pat (pat-list p) by auto
          hence wf-pat (pat-list rp) unfolding wf-pat-def wf-match-def
            unfolding pat-list-def rp tvars-match-def by force
        }
        thus ?thesis by blast
      qed
      show ∀ p ∈ set ?Q. lvar-cond-pp 0 (pat-mset-list p)
      proof
        fix rp
        assume rp ∈ set ?Q
        then obtain p where rp: rp = ?r p by auto
        show lvar-cond-pp 0 (pat-mset-list rp)
          unfolding lvar-cond-pp-def lvar-cond-def
        proof
```

```
        fix x
        assume x ∈ lvars-pp (pat-mset-list rp)
        from this[unfolded lvars-pp-def lvars-mp-def pat-mset-list-def rp]
        obtain t :: ('f, 'v) term where x ∈ vars (map-vars renVar t) by auto
        hence x ∈ range renVar by (induct t, auto)
        thus x ∈ allowed-vars 0 unfolding allowed-vars-def by auto
      qed
    qed
  qed
  also have ... = pats-complete C (pat-list ` set P)
    unfolding set-map image-comp
    unfolding Ball-image-comp o-def
  proof (intro ball-cong[OF refl])
    fix p
    assume p: p ∈ set P
    have id: pat-list (map (map (apsnd (map-vars renVar))) p) = (λ mp. apsnd
(map-vars renVar) ` mp) ` pat-list p
      unfolding pat-list-def set-map image-comp o-def ..
    note bex = bex-simps(7)
    from wf p
    have wfp: wf-pat (pat-list p)
      by (fastforce simp: subset-iff wf-pat-def wf-match-def tvars-match-def)
    from wf p
    have wf2: wf-pat (pat-list (?r p))
      by (fastforce simp: id subset-iff wf-pat-def wf-match-def tvars-match-def)
    show pat-complete C (pat-list (?r p)) = pat-complete C (pat-list p)
      apply (unfold wf-pat-complete-iff[OF wfp] wf-pat-complete-iff[OF wf2])
      apply (unfold id bex)
    proof (rule all-cong, rule bex-cong[OF refl])
      fix σ mp
    have id: map-vars renVar = (λ t. t · (Var o renVar)) using map-vars-term-eq[of
renVar] by auto
    show match-complete-wrt σ (apsnd (map-vars renVar) ` mp) = match-complete-wrt
σ mp (is ?m1 = ?m2)
      proof
        assume ?m1
        from this[unfolded match-complete-wrt-def] obtain μ
          where match: ⋀ t l. (t, l) ∈ mp ⟹ t · σ = map-vars renVar l · μ by
force
        show ?m2 unfolding match-complete-wrt-def
          by (intro exI[of - (Var o renVar) ∘s μ], insert match[unfolded id], auto)
      next
        assume ?m2
        from this[unfolded match-complete-wrt-def] obtain μ
          where match: ⋀ t l. (t, l) ∈ mp ⟹ t · σ = l · μ by force
        {
          fix t
          have t · (Var ∘ renVar) · (Var ∘ the-inv renVar) · μ = t · μ
            unfolding subst-subst
```

```
          proof (intro term-subst-eq)
            fix x
                from renaming-ass[rule-format, OF ‹improved›, unfolded renam-
ing-funs-def]
              have inj: inj renVar by auto
              from the-inv-f-f[OF this]
                 show ((Var ∘ renVar) ∘s (Var ∘ the-inv renVar) ∘s μ) x = μ x

               by (simp add: o-def subst-compose-def)
            qed
          }
          thus ?m1 unfolding match-complete-wrt-def
           by (intro exI[of - (Var o the-inv renVar) ∘s μ], insert match, auto simp:
id)
        qed
      qed
    qed
    finally show ?thesis .
  qed
qed
end
end
```

8.3 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*.

```
lemma hastype-in-map-of: distinct (map fst l) ⟹ x : σ in map-of l ⟷ (x,σ) ∈
set l
  by (auto simp: hastype-def)

lemma fun-hastype-in-map-of: distinct (map fst l) ⟹
  x : σs → τ in map-of l ⟷ ((x,σs),τ) ∈ set l
  by (auto simp: fun-hastype-def)

definition constr-list where constr-list Cs s = map fst (filter ((=) s o snd) Cs)
```

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

```
definition sorts-of-ssig-list :: (('f × 's list) × 's)list ⇒ 's list where
  sorts-of-ssig-list Cs = remdups (List.maps (λ ((f,ss),s). s # ss) Cs)

lemma sorts-of-ssig-list:
  assumes ((f,σs),τ) ∈ set Cs
  shows set σs ⊆ set (sorts-of-ssig-list Cs) τ ∈ set (sorts-of-ssig-list Cs)
  using assms
  by (auto simp: sorts-of-ssig-list-def List.maps-def)
```

definition *max-arity-list* **where**
max-arity-list Cs = *max-list* (*map* (*length o snd o fst*) *Cs*)

lemma *max-arity-list*:
((*f*,σ*s*),τ) ∈ *set Cs* ⟹ *length* σ*s* ≤ *max-arity-list Cs*
by (*force simp*: *max-arity-list-def o-def intro*! :*max-list*)

locale *pattern-completeness-list* =
fixes *Cs*
assumes *dist*: *distinct* (*map fst Cs*)
and *inhabited*: *decide-nonempty-sorts* (*sorts-of-ssig-list Cs*) *Cs* = *None*
begin

lemma *nonempty-sort*: ⋀ σ. σ ∈ *set* (*sorts-of-ssig-list Cs*) ⟹ ¬ *empty-sort* (*map-of Cs*) σ
using *decide-nonempty-sorts*(*1*)[*OF dist inhabited*]
by (*auto elim*: *not-empty-sortE*)

lemma *compute-inf-sorts*: σ ∈ *compute-inf-sorts Cs* ⟷ ¬ *finite-sort* (*map-of Cs*) σ
apply (*subst compute-inf-sorts*(*2*)[*OF - dist*])
using *nonempty-sort*
by (*auto intro*!:*nonempty-sort simp*: *fun-hastype-in-map-of*[*OF dist*] *dest*!: *sorts-of-ssig-list*(*1*))

lemma *compute-card-sorts*: *snd* (*compute-inf-card-sorts Cs*) = *card-of-sort* (*map-of Cs*)
apply (*rule compute-inf-card-sorts*(*2*)[*OF refl - dist surjective-pairing*])
by (*auto intro*!: *nonempty-sort simp*: *fun-hastype-in-map-of*[*OF dist*] *dest*!: *sorts-of-ssig-list*(*1*))

sublocale *pattern-completeness-context-with-assms*
improved set (*sorts-of-ssig-list Cs*) *map-of Cs max-arity-list Cs constr-list Cs*
λ *s*. *s* ∈ *compute-inf-sorts Cs*
snd (*compute-inf-card-sorts Cs*)
for *improved*
proof
{
fix *f ss s*
assume *f* : *ss* → *s in map-of Cs*
hence ((*f*,*ss*),*s*) ∈ *set Cs* **by** (*auto dest*!: *fun-hastypeD map-of-SomeD*)
from *sorts-of-ssig-list*[*OF this*] *max-arity-list*[*OF this*]
show *insert s* (*set ss*) ⊆ *set* (*sorts-of-ssig-list Cs*) *length ss* ≤ *max-arity-list Cs*
by *auto*
}
show *finite* (*dom* (*map-of Cs*)) **by** (*auto simp*: *finite-dom-map-of*)
show *set* (*constr-list Cs s*) = {(*f*,*ss*). *f* : *ss* → *s in map-of Cs*} **for** *s*
unfolding *constr-list-def set-map o-def* **using** *dist*
by (*force simp*: *fun-hastype-def*)
{
fix *f ss s*

assume $(f,ss) \in set\ (constr\text{-}list\ Cs\ s)$
hence $((f,ss),s) \in set\ Cs$ **unfolding** *constr-list-def* **by** *auto*
from *max-arity-list*[*OF this*] **have** $length\ ss \leq max\text{-}arity\text{-}list\ Cs$ **by** *auto*
}
then show m: $\forall a{\in}length\ `\ snd\ `\ set\ (constr\text{-}list\ Cs\ s).\ a \leq max\text{-}arity\text{-}list\ Cs$ **for** s **by** *auto*
qed (*auto simp*: *compute-inf-sorts nonempty-sort compute-card-sorts*)

thm *pat-complete-impl*
thm *pat-complete-lin-impl*

end

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

Finally: a pattern completeness decision procedure for arbitrary inputs, assuming sensible inputs; this is the old decision procedure

context
fixes m :: *nat* — upper bound on arities of constructors
and $Cl :: 's \Rightarrow ('f \times 's\ list)list$ — a function to compute all constructors of given sort as list
and $Is :: 's \Rightarrow bool$ — a function to indicate whether a sort is infinite
and $Cd :: 's \Rightarrow nat$ — a function to compute finite cardinality of sort
begin

definition *pat-complete-impl-old* = *pattern-completeness-context.pat-complete-impl m Cl Is Cd False undefined undefined undefined undefined*
definition *pats-impl-old* = *pattern-completeness-context.pats-impl m Cl Is Cd False undefined undefined undefined*
definition *pat-impl-old* = *pattern-completeness-context.pat-impl m Cl Is False undefined undefined*
definition *pat-inner-impl-old* = *pattern-completeness-context.pat-inner-impl Is False undefined*
definition *match-decomp'-impl-old* = *pattern-completeness-context.match-decomp'-impl Is False undefined*

definition *find-var-old* :: $('f,'v,'s)$*match-problem-lr list* ⇒ - **where**
find-var-old p = (*case List.maps* (λ (*lx*,-). *lx*) *p of*
(*x*,*t*) # - ⇒ *x*
| [] ⇒ (*let* (-,*rx*,*b*) = *hd p*
in case hd rx of (*x*, *s* # *t* # -) ⇒ *hd* (*the* (*conflicts s t*))))

lemma *find-var-old*: *find-var False p* = *Some* (*find-var-old p*)
unfolding *find-var-old-def find-var-def if-False* **by** (*auto split*: *list.splits*)

lemmas *pat-complete-impl-old-code*[*code*] = *pattern-completeness-context.pat-complete-impl-def*[*of m Cl Is Cd False undefined undefined undefined undefined*,
folded pat-complete-impl-old-def pats-impl-old-def,

unfolded if-False Let-def]

private lemma *triv-ident*: *False* ∧ *x* ⟷ *False True* ∧ *x* ⟷ *x* **by** *auto*

lemmas *pat-impl-old-code*[*code*] = *pattern-completeness-context.pat-impl-def*[*of m Cl Is False undefined undefined*,
folded pat-impl-old-def pat-inner-impl-old-def,
unfolded find-var-old option.simps triv-ident if-False]

lemma *pats-impl-old-code*[*code*]:
pats-impl-old n nl ps =
(*case ps of* [] ⇒ *True*
| *p* # *ps1* ⇒
(*case pat-impl-old n nl p of Incomplete* ⇒ *False*
| *New-Problems* (*n'*, *nl'*, *ps2*) ⇒ *pats-impl-old n' nl'* (*ps2* @ *ps1*)))
unfolding *pats-impl-old-def pattern-completeness-context.pats-impl.simps*[*of - - - - - - - - - ps*]
unfolding *pat-impl-old-def*[*symmetric*]
unfolding *pat-impl-old-code*
by (*auto split*: *list.splits option.splits*)

lemmas *match-decomp'-impl-old-code*[*code*] =
pattern-completeness-context.match-decomp'-impl-def[*of Is False undefined*, *folded match-decomp'-impl-old-def*,
unfolded pattern-completeness-context.apply-decompose'-def triv-ident if-False]

lemmas *pat-inner-impl-old-code*[*code*] =
pattern-completeness-context.pat-inner-impl.simps[*of Is False undefined*, *folded pat-inner-impl-old-def match-decomp'-impl-old-def*]

context
fixes
C :: ('*f* × '*s list*) ⇒ '*s option*
and *rn* :: *nat* ⇒ '*v*
and *rv* :: '*v* ⇒ '*v*
and *fidl-solver* :: ((*nat*×'*s*) × *int*)*list* × ((*nat*×'*s*) × (*nat*×'*s*))*list list* ⇒ *bool*
begin
definition *pat-complete-impl-new* = *pattern-completeness-context.pat-complete-impl m Cl Is Cd True C rn rv fidl-solver*
definition *pats-impl-new* = *pattern-completeness-context.pats-impl m Cl Is Cd True C rn fidl-solver*
definition *pat-impl-new* = *pattern-completeness-context.pat-impl m Cl Is True C rn*
definition *pat-inner-impl-new* = *pattern-completeness-context.pat-inner-impl Is True rn*
definition *match-decomp'-impl-new* = *pattern-completeness-context.match-decomp'-impl Is True rn*
definition *find-var-new* = *find-var True*

lemmas *pat-complete-impl-new-code*[*code*] = *pattern-completeness-context.pat-complete-impl-def*[*of m Cl Is Cd True C rn rv fidl-solver*,
folded pat-complete-impl-new-def pats-impl-new-def,
unfolded if-True Let-def]

lemmas *pat-impl-new-code*[*code*] = *pattern-completeness-context.pat-impl-def*[*of m Cl Is True C rn*,
folded pat-impl-new-def pat-inner-impl-new-def find-var-new-def,
unfolded triv-ident]

lemmas *pats-impl-new-code*[*code*] = *pattern-completeness-context.pats-impl.simps*[*of m Cl Is Cd True C rn fidl-solver*,
folded pats-impl-new-def pat-impl-new-def]

lemmas *match-decomp'-impl-new-code*[*code*] =
pattern-completeness-context.match-decomp'-impl-def[*of Is True rn*,
folded match-decomp'-impl-new-def,
unfolded pattern-completeness-context.apply-decompose'-def triv-ident]

lemmas *pat-inner-impl-new-code*[*code*] =
pattern-completeness-context.pat-inner-impl.simps[*of Is True rn*,
folded pat-inner-impl-new-def match-decomp'-impl-new-def]

lemmas *find-var-new-code*[*code*] =
find-var-def[*of True*,
folded find-var-new-def,
unfolded if-True]

end
end

definition *decide-pat-complete* :: ((*'f* × *'s list*) × *'s*)*list* ⇒ (*'f*,*'v*,*'s*)*pats-problem-list* ⇒ *bool* **where**
decide-pat-complete Cs P = (*let*
m = *max-arity-list Cs*;
Cl = *constr-list Cs*;
(*IS*,*CD*) = *compute-inf-card-sorts Cs*
in pat-complete-impl-old m Cl (λ *s*. *s* ∈ *IS*) *CD*) *P*

definition *decide-pat-complete-lin* :: ((*'f* × *'s list*) × *'s*)*list* ⇒ (*'f*,*'v*,*'s*)*pats-problem-list* ⇒ *bool* **where**
decide-pat-complete-lin Cs P = (*let*
m = *max-arity-list Cs*;
Cl = *constr-list Cs*
in pattern-completeness-context.pat-complete-lin-impl m Cl P)

theorem *decide-pat-complete-lin*:

assumes *dist*: *distinct* (*map fst Cs*)
and *non-empty-sorts*: *decide-nonempty-sorts* (*sorts-of-ssig-list Cs*) *Cs* = *None*
and *P*: *snd* ‘ $\bigcup$ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat P*))) $\subseteq$ *set* (*sorts-of-ssig-list Cs*)
and *left-linear*: *Ball* (*set P*) *ll-pp*
shows *decide-pat-complete-lin Cs P* = *pats-complete* (*map-of Cs*) (*pat-list* ‘ *set P*)
proof−
interpret *pattern-completeness-list Cs*
apply *unfold-locales*
using *dist non-empty-sorts*.
show *?thesis*
unfolding *decide-pat-complete-lin-def Let-def*
by (*rule pat-complete-lin-impl*[*OF P left-linear*])
qed

theorem *decide-pat-complete*:
assumes *dist*: *distinct* (*map fst Cs*)
and *non-empty-sorts*: *decide-nonempty-sorts* (*sorts-of-ssig-list Cs*) *Cs* = *None*
and *P*: *snd* ‘ $\bigcup$ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat P*))) $\subseteq$ *set* (*sorts-of-ssig-list Cs*)
shows *decide-pat-complete Cs P* = *pats-complete* (*map-of Cs*) (*pat-list* ‘ *set P*)
proof−
interpret *pattern-completeness-list Cs*
apply *unfold-locales*
using *dist non-empty-sorts*.
show *?thesis*
unfolding *decide-pat-complete-def Let-def pat-complete-impl-old-def*
apply (*unfold case-prod-beta*)
apply (*rule pat-complete-impl*[*OF - - - P*]) **by** *auto*
qed

definition *decide-pat-complete-fidl* :: - ⇒ - ⇒ - ⇒ ((*'f* × *'s list*) × *'s*)*list* ⇒ (*'f*,*'v*,*'s*)*pats-problem-list* ⇒ *bool* **where**
decide-pat-complete-fidl rn rv idl Cs P = (*let*
m = *max-arity-list Cs*;
Cl = *constr-list Cs*;
Cm = *Mapping.of-alist Cs*;
(*IS*,*CD*) = *compute-inf-card-sorts Cs*
in pat-complete-impl-new m Cl (λ *s*. *s* ∈ *IS*) *CD* (*Mapping.lookup Cm*)) *rn rv idl P*

definition *fvf-pp-list pp* =
[[*y*. (*t'*, *Var y*) ← *pp*, *t'* = *t*]. *t* ← *remdups* (*map fst pp*)]

theorem *decide-pat-complete-fidl*:

assumes *dist*: *distinct* (*map fst Cs*)
and *non-empty-sorts*: *decide-nonempty-sorts* (*sorts-of-ssig-list Cs*) *Cs* = *None*
and *P*: *snd* ‘ $\bigcup$ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat P*))) $\subseteq$ *set* (*sorts-of-ssig-list Cs*)
and *ren*: *renaming-funs rn rv*
and *fidl-solver*: *finite-idl-solver fidl-solver*
shows *decide-pat-complete-fidl rn rv fidl-solver Cs P* $\longleftrightarrow$ *pats-complete* (*map-of Cs*) (*pat-list* ‘ *set P*)
(**is** *?l* $\longleftrightarrow$ *?r*)
proof −
interpret *pattern-completeness-list Cs*
apply *unfold-locales*
using *dist non-empty-sorts*.
have *nemp*:
$\forall$ *f* τs τ τ'. *f* : $\tau s \to \tau$ *in map-of Cs* $\longrightarrow$ $\tau' \in$ *set* τs $\longrightarrow$ ¬ *empty-sort* (*map-of Cs*) τ'
using *C-sub-S* **by** (*auto intro*!: *nonempty-sort*)
obtain *inf cd* **where** *compute-inf-card-sorts Cs* = (*inf*,*cd*) **by** *force*
with *compute-inf-card-sorts*(*2*,*3*)[*OF refl nemp dist this*]
have *cics*: *compute-inf-card-sorts Cs* = (*compute-inf-sorts Cs*,*card-of-sort* (*map-of Cs*))
by *auto*
have *Cm*: *Mapping.lookup* (*Mapping.of-alist Cs*) = *map-of Cs* **using** *dist*
using *lookup-of-alist* **by** *fastforce*
show *?thesis*
apply (*unfold decide-pat-complete-fidl-def Let-def case-prod-beta*)
unfolding *pat-complete-impl-new-def*
using *pat-complete-impl*[*OF Cm ren fidl-solver P*]
by *auto*
qed

export-code *decide-pat-complete-lin* **checking**
export-code *decide-pat-complete* **checking**
export-code *decide-pat-complete-fidl* **checking**

end

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

abbreviation *basic-ground-terms* :: $('f,'s)ssig \Rightarrow ('f,'s)ssig \Rightarrow ('f,unit)term\ set$ ($\langle\mathcal{B}'(-,-')\rangle$) **where**
$\mathcal{B}(C,D) \equiv \mathcal{B}(C,D,\lambda x.\ None)$

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

lemma *matches-subst*: $l\ matches\ t \Longrightarrow l\ matches\ t{\cdot}\sigma$
by (*auto simp*: *matches-def simp flip*: *subst-subst-compose*)

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

lemma *pat-complete-lhssD*:
assumes *comp*: *pat-complete-lhss C D L* **and** *t*: $t \in \mathcal{B}(C,D,\emptyset)$
shows $\exists\, l \in L.\ l\ matches\ t$
proof −
note $* = map\text{-}subst\text{-}hastype[OF\ sorted\text{-}map\text{-}empty,\ of\ C\ -\ -\ \emptyset{::}unit{\rightharpoonup}-\ undefined]$
from *t* **have** $t{\cdot}undefined \in \mathcal{B}(C,D)$ (**is** $?t \in$ -)
by (*force simp*: *basic-terms-def* $*$ *cong*: *ex-cong*)
from *comp*[*unfolded pat-complete-lhss-def*, *rule-format*, *OF this*]
obtain *l* **where** *l*: $l \in L\ l\ matches\ ?t$ **by** *auto*
from *t*
have *t2*: $?t \cdot undefined = t$
by (*auto simp*: *basic-terms-def o-def*
simp: *hastype-in-Term-empty-imp-map-subst-subst hastype-in-Term-empty-imp-map-subst-id*)
from *l* **show** $\exists\, l \in L.\ l\ matches\ t$
apply (*subst t2*[*symmetric*])
by (*force simp*: *matches-subst*)
qed

definition *pats-of-lhss* :: $((('f \times 's\ list) \times 's)list \Rightarrow ('f,'v)term\ list \Rightarrow ('f,'v,'s)pat\text{-}problem\text{-}list\ list$ **where**
$pats\text{-}of\text{-}lhss\ D\ lhss = (let\ pats = [Fun\ f\ (map\ Var\ (zip\ [0..<length\ ss]\ ss)).\ ((f,ss),s) \leftarrow D]$
$in\ [[[(pat,lhs)].\ lhs \leftarrow lhss].\ pat \leftarrow pats])$

definition *check-signatures* :: $((('f \times 's\ list) \times 's)list \Rightarrow ((('f \times 's\ list) \times 's)list \Rightarrow showsl\ check$ **where**
check-signatures C D = *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 ''a defined symbol has argument sort that is not known in
constructors'')) ss) D;
    (case (decide-nonempty-sorts S C) of None ⇒ return () | Some s ⇒ error
(showsl-lit (STR ''some sort is empty'')))
  }

definition decide-pat-complete-linear-lhss ::
  (('f × 's list) × 's)list ⇒ (('f × 's list) × 's)list ⇒ ('f,'v)term list ⇒ showsl +
bool where
  decide-pat-complete-linear-lhss C D lhss = do {
    check-signatures C D;
    return (decide-pat-complete-lin C (pats-of-lhss D lhss))
  }

definition decide-pat-complete-lhss ::
  (('f × 's list) × 's)list ⇒ (('f × 's list) × 's)list ⇒ ('f,'v)term list ⇒ showsl +
bool where
  decide-pat-complete-lhss C D lhss = do {
    check-signatures C D;
    return (decide-pat-complete C (pats-of-lhss D lhss))
  }

definition decide-pat-complete-lhss-fidl ::
  - ⇒ - ⇒ - ⇒ (('f × 's list) × 's)list ⇒ (('f × 's list) × 's)list ⇒ ('f,'v)term list
⇒ showsl + bool where
  decide-pat-complete-lhss-fidl rn rv fidl-solver C D lhss = do {
    check-signatures C D;
    return (decide-pat-complete-fidl rn rv fidl-solver C (pats-of-lhss D lhss))
  }

lemma pats-of-lhss-vars: assumes condD: ∀ x∈set D. ∀ a b. (∀ x2. x ≠ ((a, b),
x2)) ∨ (∀ x∈set b. x ∈ S)
  shows snd ` ⋃ (vars ` fst ` set (concat (concat (pats-of-lhss D lhss)))) ⊆ S
proof −
  {
    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 condD
    have si ∈ S by auto
  }
  thus ?thesis unfolding pats-of-lhss-def by force
qed

lemma check-signatures: assumes isOK(check-signatures C D)
```

shows *distinct* (*map fst C*) (**is** *?G1*)
and *distinct* (*map fst D*) (**is** *?G2*)
and $\forall x{\in}set\ D.\ \forall a\ b.\ (\forall x2.\ x \neq ((a,\ b),\ x2)) \vee (\forall x{\in}set\ b.\ x \in set\ (sorts\text{-}of\text{-}ssig\text{-}list\ C))$ (**is** *?G3*)
and *decide-nonempty-sorts* (*sorts-of-ssig-list C*) *C* = *None* (**is** *?G4*)
proof −
let *?C* = *map-of C*
let *?D* = *map-of D*
define *S* **where** *S* = *sorts-of-ssig-list C*
have *dist*: *distinct* (*map fst C*) **and** *distD*: *distinct* (*map fst D*)
and *dec*: *decide-nonempty-sorts S C* = *None*
and *condD*: $\forall x{\in}set\ D.\ \forall a\ b.\ (\forall x2.\ x \neq ((a,\ b),\ x2)) \vee (\forall x{\in}set\ b.\ x \in set\ S)$
using *assms*
apply (*unfold check-signatures-def*)
apply (*unfold Let-def S-def*[*symmetric*])
apply (*auto split*: *prod.splits option.splits*)
done
show *?G1 ?G2 ?G3 ?G4* **unfolding** *S-def*[*symmetric*] **by** *fact*+
qed

lemma *pats-of-lhss*:
assumes *isOK*(*check-signatures C D*)
shows *pats-complete* (*map-of C*) (*pat-list* ‘ *set* (*pats-of-lhss D lhss*)) =
$(\forall t{\in}\mathcal{B}(map\text{-}of\ C,map\text{-}of\ D).\ \exists l{\in}set\ lhss.\ l\ matches\ t)$
proof −
define *S* **where** *S* = *sorts-of-ssig-list C*
note ∗ = *check-signatures*[*OF assms*, *folded S-def*]
note *distC* = ∗(*1*) **note** *distD* = ∗(*2*) **note** *condD* = ∗(*3*) **note** *dec* = ∗(*4*)
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*
note *condD* = *condD*[*folded S-def*]
note *dec* = *dec*[*folded S-def*]
let *?C* = *map-of C*
let *?D* = *map-of D*
let *?L* = $\{\ pat \cdot \sigma \mid pat\ \sigma.\ pat \in set\ pats \wedge \sigma :_s \{x : \iota\ in\ \mathcal{V}.\ \iota \in set\ S\} \to \mathcal{T}(?C)\}$

interpret *pattern-completeness-list C*
rewrites *sorts-of-ssig-list C* = *S*
apply *unfold-locales*
using *distC dec* **by** (*auto simp*: *S-def*)
from *condD*
have *wf*: *wf-pats* (*pat-list* ‘ *set P*)
by (*force simp*: *P-def pats-def wf-pats-def wf-pat-def pat-list-def wf-match-def tvars-match-def*
elim!: *in-set-zipE*)
let *?match-lhs* = λ*t*. ∃ *l* ∈ *set lhss*. *l matches t*
have *pats-complete ?C* (*pat-list* ‘ *set* (*pats-of-lhss D lhss*))
= *pats-complete ?C* (*pat-list* ‘ *set P*) **unfolding** *P-def pats-of-lhss-def pats-def*

by *auto*
also note *wf-pats-complete-iff*[*OF wf*]
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 (∀ *f* :$_s$ {*x* : ι *in* $\mathcal{V}$. ι ∈ *set S*} → $\mathcal{T}$(*map-of C*).
∀ *pp*∈{{{(*pat*, *lhs*)} |*lhs*. *lhs* ∈ *set lhss*} |*pat*. *pat* ∈ *set pats*}.
∃ *mp*∈*pp*. *match-complete-wrt f mp*) = *Ball* { *pat* · σ | *pat* σ. *pat* ∈ *set pats* ∧ σ :$_s$ {*x* : ι *in* $\mathcal{V}$. ι ∈ *set S*} → $\mathcal{T}$(*?C*)} *?match-lhs* (**is** - = *Ball ?L* -)
apply (*simp add*: *imp-ex match-complete-wrt-def matches-def Bex-def conj-commute*
imp-conjL flip:*ex-simps*(*1*) *all-simps*(*6*) *split*: *prod.splits*
cong: *all-cong1 ex-cong1 conj-cong imp-cong*)
apply (*subst all-comm*)
by (*simp add*: *ac-simps verit-bool-simplify*(*4*) *o-def*)
also have *?L* = $\mathcal{B}$(*?C*,*?D*,∅) (**is** - = *?R*)
proof
{
fix *pat* **and** σ
assume *pat*: *pat* ∈ *set pats* **and** *subst*: σ :$_s$ {*x* : ι *in* $\mathcal{V}$. ι ∈ *set S*} → $\mathcal{T}$(*?C*)
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*) ∈ *set D* **by** *auto*
from *distD inDs* **have** *f*: *f* : *ss* → *s in ?D* **unfolding** *fun-hastype-def* **by** *simp*
{
fix *i*
assume *i*: *i* < *length ss*
hence *ss* ! *i* ∈ *set ss* **by** *auto*
with *inDs condD* **have** *ss* ! *i* ∈ *set S* **by** (*auto simp*: *S-def*)
then
have σ (*i*, *ss* ! *i*) : *ss* ! *i in* $\mathcal{T}$(*?C*)
by (*auto intro*!: *sorted-mapD*[*OF subst*] *simp*: *hastype-restrict*)
} **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*) **unfolding** *list-all2-conv-all-nth ts-def* **using** *ssigma* **by** *auto*
have *pat*: *pat* · σ = *Fun f ts*
unfolding *pat ts-def* **by** (*auto intro*: *nth-equalityI*)
from *pat f ts* **have** *pat* · σ ∈ *?R* **unfolding** *basic-terms-def* **by** *auto*
}
thus *?L* ⊆ *?R* **by** *auto*
{
fix *f ss s* **and** *ts*
assume *f*: *f* : *ss* → *s in ?D* **and** *ts*: *ts* :$_l$ *ss in* $\mathcal{T}$(*?C*)
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*) ∈ *set D* **unfolding** *fun-hastype-def* **by** (*metis map-of-SomeD*)
hence *pat*: *pat* ∈ *set pats* **unfolding** *pat-def pats-def* **by** *force*
define σ **where** σ *x* = (*case x of* (*i*,*s*) ⇒ *if i* < *length ss* ∧ *s* = *ss* ! *i then*

```
ts ! i else
      (SOME t. t : s in 𝒯(?C))) for x
    have id: Fun f ts = pat · σ unfolding pat-def using len
      by (auto intro!: nth-equalityI simp: σ-def)
    have ssigma: σ :ₛ {x : ι in 𝒱. ι ∈ set S} → 𝒯(?C)
    proof (intro sorted-mapI)
      fix x ι
      assume x : ι in {x : ι in 𝒱. ι ∈ set S}
      then have ι = snd x and s: ι ∈ set S by auto
      then obtain i where x: x = (i,ι) by (cases x, auto)
      show σ x : ι in 𝒯(?C)
      proof (cases i < length ss ∧ ι = 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
          by (auto simp add: list-all2-conv-all-nth)
      next
        case False
        hence id: σ x = (SOME t. t : ι in 𝒯(?C)) unfolding x σ-def by auto
        from decide-nonempty-sorts(1)[OF distC dec] s
        have ∃ t. t : ι in 𝒯(?C) by (auto elim!: not-empty-sortE simp: S-def)
        from someI-ex[OF this] have σ x : ι in 𝒯(?C) 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 .
qed

theorem decide-pat-complete-lhss:
 fixes C D :: (('f × 's list) × 's) list and lhss :: ('f,'v)term list
 assumes decide-pat-complete-lhss C D lhss = return b
 shows b = pat-complete-lhss (map-of C) (map-of D) (set lhss)
proof −
 let ?C = map-of C
 let ?D = map-of D
 define S where S = sorts-of-ssig-list C
 define P where P = pats-of-lhss D lhss
 have sig: isOK(check-signatures C D)
   and b: b = decide-pat-complete C P
   using assms
   apply (unfold decide-pat-complete-lhss-def)
   apply (unfold Let-def P-def[symmetric] S-def[symmetric])
   by auto
 note ∗ = check-signatures[OF sig]
```

note *distC* = ∗(*1*) **note** *distD* = ∗(*2*) **note** *condD* = ∗(*3*) **note** *dec* = ∗(*4*)
interpret *pattern-completeness-list C*
rewrites *sorts-of-ssig-list C* = *S*
apply *unfold-locales*
using ∗ **by** (*auto simp*: *S-def*)
have *b* = *pats-complete ?C* (*pat-list* ‘ *set P*)
apply (*unfold b*)
apply (*rule decide-pat-complete*[*OF distC dec*[*unfolded S-def*]])
apply (*unfold P-def*)
apply (*rule pats-of-lhss-vars*[*OF condD*[*unfolded P-def S-def*]])
done
also have ... = (∀ *t*∈*B*(*?C*,*?D*). ∃ *l*∈*set lhss*. *l matches t*) **unfolding** *P-def*
by (*rule pats-of-lhss*[*OF sig*])
finally show *?thesis* **unfolding** *pat-complete-lhss-def* **.**
qed

theorem *decide-pat-complete-linear-lhss*:
fixes *C D* :: ((′*f* × ′*s list*) × ′*s*) *list* **and** *lhss* :: (′*f*,′*v*)*term list*
assumes *decide-pat-complete-linear-lhss C D lhss* = *return b*
and *linear*: *Ball* (*set lhss*) *linear-term*
shows *b* = *pat-complete-lhss* (*map-of C*) (*map-of D*) (*set lhss*)
proof −
let *?C* = *map-of C*
let *?D* = *map-of D*
define *S* **where** *S* = *sorts-of-ssig-list C*
define *P* **where** *P* = *pats-of-lhss D lhss*
have *sig*: *isOK*(*check-signatures C D*)
and *b*: *b* = *decide-pat-complete-lin C P*
using *assms*
apply (*unfold decide-pat-complete-linear-lhss-def*)
apply (*unfold Let-def P-def*[*symmetric*] *S-def*[*symmetric*])
by *auto*
note ∗ = *check-signatures*[*OF sig*]
note *distC* = ∗(*1*) **note** *distD* = ∗(*2*) **note** *condD* = ∗(*3*) **note** *dec* = ∗(*4*)
interpret *pattern-completeness-list C*
rewrites *sorts-of-ssig-list C* = *S*
apply *unfold-locales*
using ∗ **by** (*auto simp*: *S-def*)
have *b* = *pats-complete ?C* (*pat-list* ‘ *set P*)
apply (*unfold b*)
apply (*rule decide-pat-complete-lin*[*OF distC dec*[*unfolded S-def*]])
apply (*unfold P-def*)
apply (*rule pats-of-lhss-vars*[*OF condD*[*unfolded P-def S-def*]])
apply (*fold P-def*)
proof −
show *Ball* (*set P*) *ll-pp* **unfolding** *ll-pp-def*
proof (*intro ballI*)
fix *p mp*
assume *p* ∈ *set P* **and** *mp*: *mp* ∈ *set p*

```
    from this[unfolded P-def pats-of-lhss-def, simplified]
    obtain pat where p: p = map (λlhs. [(pat, lhs)]) lhss by auto
    from mp[unfolded p, simplified] obtain l where mp: mp = [(pat, l)]
      and l: l ∈ set lhss by auto
    have vars: vars-mp-mset (mp-list mp) = vars-term-ms l
      unfolding mp vars-mp-mset-def by auto
    from l linear have l: linear-term l by auto
    hence dist: distinct (vars-term-list l) by (rule linear-term-distinct-vars)
    have id: vars-term-ms l = mset (vars-term-list l)
    proof (induct l)
      case (Fun f ts)
      thus ?case by (simp add: vars-term-list.simps, induct ts, auto)
    qed (auto simp: vars-term-list.simps)
    show ll-mp (mp-list mp) unfolding ll-mp-def vars id using dist
      by (simp add: distinct-count-atmost-1)
  qed
 qed
 also have ... = (∀ t∈B(?C,?D). ∃ l∈set lhss. l matches t) unfolding P-def
  by (rule pats-of-lhss[OF sig])
 finally show ?thesis unfolding pat-complete-lhss-def .
qed

theorem decide-pat-complete-lhss-fidl:
 fixes C D :: (('f × 's list) × 's) list and lhss :: ('f,'v)term list
 assumes decide-pat-complete-lhss-fidl rn rv fidl-solver C D lhss = return b
  and ren: renaming-funs rn rv
  and idl: finite-idl-solver fidl-solver
 shows b = pat-complete-lhss (map-of C) (map-of D) (set lhss)
proof −
 let ?C = map-of C
 let ?D = map-of D
 define S where S = sorts-of-ssig-list C
 define P where P = pats-of-lhss D lhss
 have sig: isOK(check-signatures C D)
  and b: b = decide-pat-complete-fidl rn rv fidl-solver C P
  using assms
   apply (unfold decide-pat-complete-lhss-fidl-def)
   apply (unfold Let-def P-def[symmetric] S-def[symmetric])
  by auto
 note ∗ = check-signatures[OF sig]
 note distC = ∗(1) note distD = ∗(2) note condD = ∗(3) note dec = ∗(4)
 interpret pattern-completeness-list C
  rewrites sorts-of-ssig-list C = S
   apply unfold-locales
  using ∗ by (auto simp: S-def)
 have b = pats-complete ?C (pat-list ‘ set P)
  apply (unfold b)
  apply (rule decide-pat-complete-fidl[OF distC dec[unfolded S-def] - ren idl])
  apply (unfold P-def)
```

apply (*rule pats-of-lhss-vars*[*OF condD*[*unfolded P-def S-def*]])
done
also have ... = ($\forall\, t{\in}\mathcal{B}(?C,?D).\ \exists\, l{\in}$*set lhss*. *l matches t*) **unfolding** *P-def*
by (*rule pats-of-lhss*[*OF sig*])
finally show *?thesis* **unfolding** *pat-complete-lhss-def* **.**
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*,∅::*unit*⇀'*s*). ∃ *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-signatures C D;
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*:
fixes *C D* :: (('*f* × '*s list*) × '*s*) *list* **and** *lhss* :: ('*f*,'*v*)*term list*
assumes *decide-strong-quasi-reducible C D lhss* = *return b*
shows *b* = *strong-quasi-reducible* (*map-of C*) (*map-of D*) (*set lhss*)
proof −
let *?C* = *map-of C*
let *?D* = *map-of D*
let *?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*
have *pats*: *patL* ∈ *set pats* ⟷ (∃ ((*f*,*ss*),*s*) ∈ *set D*. *patL* = *term-and-args f* (*map Var* (*zip* [*0*..<*length ss*] *ss*)))
for *patL*
by (*force simp*: *pats-def split*: *prod.splits*)
define *P* **where** *P* = *map* (*List.maps* (λ *pat*. *map* (λ *lhs*. [(*pat*,*lhs*)]) *lhss*)) *pats*
define *V* **where** *V* = {*x* : ι *in* $\mathcal{V}$. ι ∈ *set* (*sorts-of-ssig-list C*)}
let *?match-lhs* = λ*t*. ∃ *l* ∈ *set lhss*. *l matches t*
from *assms*(*1*)
have *b*: *b* = *decide-pat-complete C P*
and *sig*: *isOK* (*check-signatures C D*)

by (*auto simp*: *decide-strong-quasi-reducible-def pats-def*[*symmetric*] *Let-def P-def*[*symmetric*]
split: *prod.splits option.splits*)
note ∗ = *check-signatures*[*OF sig*]
note *distC* = ∗(*1*) **note** *distD* = ∗(*2*) **note** *condD* = ∗(*3*) **note** *dec* = ∗(*4*)
interpret *pattern-completeness-list C*
apply *unfold-locales* **using** *distC dec*.
have *wf*: *wf-pats* (*pat-list* ‘ *set P*) **using** *condD*
by (*force simp*: *P-def pats-def wf-pats-def wf-pat-def pat-list-def wf-match-def tvars-match-def*
term-and-args-def List.maps-def
elim!: *in-set-zipE split*: *prod.splits*)
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*+
have *b* = *pats-complete ?C* (*pat-list* ‘ *set P*)
apply (*unfold b*)
proof (*rule decide-pat-complete*[*OF dist*(*1*) *dec*])
{
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 condD*
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
also have . . . ⟷
(∀ *σ* :$_s$ *V* → $\mathcal{T}$(*?C*). ∀ *patL* ∈ *set pats*. (∃ *pat* ∈ *set patL*. *?match-lhs* (*pat* · *σ*))) (**is** - ⟷ *?L*)
apply (*unfold wf-pats-complete-iff*[*OF wf*])
apply (*fold V-def*)
apply (*unfold* ∗)
apply (*simp add*: *imp-ex match-complete-wrt-def matches-def flip*: *Ball-def*)
apply (*rule all-cong*)
apply (*rule ball-cong*)
apply *simp*
apply (*auto simp*: *pats*)
by *blast*
also have . . . ⟷
(∀ *f ss s ts*. *f* : *ss* → *s in ?D* ⟶ *ts* :$_l$ *ss in* $\mathcal{T}$(*?C*) ⟶
(∃ *ti* ∈ *set* (*term-and-args f ts*). *?match-lhs ti*)) (**is** - = *?R*)
proof (*intro iffI allI ballI impI*)
fix *patL* **and** *σ*
assume *patL*: *patL* ∈ *set pats* **and** *subst*: *σ* :$_s$ *V* → $\mathcal{T}$(*?C*) **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) ∈ set D by auto
    from distD inDs have f: f : ss → s in ?D unfolding fun-hastype-def by simp
    {
      fix i
      assume i: i < length ss
      hence ss ! i ∈ set ss by auto
      with inDs condD have ss ! i ∈ set ?S by auto
      then have σ (i, ss ! i) : ss ! i in 𝒯(?C)
        by (auto intro!: sorted-mapD[OF subst] simp: V-def)
    } note ssigma = this
    define ts where ts = (map (λ i. σ (i, ss ! i)) [0..<length ss])
    have ts: ts :l ss in 𝒯(?C) unfolding list-all2-conv-all-nth ts-def using ssigma
by auto
    from R[rule-format, OF f ts] obtain ti where ti: ti ∈ set (term-and-args f ts)
and match: ?match-lhs ti by auto
   have map (λ pat. pat · 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 ∃ pat∈set patL. ?match-lhs (pat · σ) by auto
  next
    fix f ss s ts
    assume f: f : ss → s in ?D and ts: ts :l ss in 𝒯(?C) 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) ∈ set D unfolding fun-hastype-def by (metis map-of-SomeD)
    hence patL: patL ∈ set pats unfolding patL-def pats-def by force
    define σ where σ x = (case x of (i,s) ⇒ if i < length ss ∧ s = ss ! i then ts
! i else
      (SOME t. t : s in 𝒯(?C))) for x
    have ssigma: σ :s V → 𝒯(?C)
    proof (intro sorted-mapI)
      fix x s
      assume x : s in V
      then obtain i where x: x = (i,s) and s: s ∈ set ?S by (cases x, auto simp:
V-def)
      show σ x : s in 𝒯(?C)
      proof (cases i < length ss ∧ 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
          by (simp add: list-all2-conv-all-nth)
      next
        case False
        hence id: σ x = (SOME t. t : s in 𝒯(?C)) unfolding x σ-def by auto
        from decide-nonempty-sorts(1)[OF dist dec, rule-format, OF s]
        have ∃ t. t : s in 𝒯(?C) by (auto elim!: not-empty-sortE)
        from someI-ex[OF this] have σ x : s in 𝒯(?C,∅) unfolding id .
```

thus *?thesis* **unfolding** *x* **by** *auto*
qed
qed
from *L*[*rule-format*, *OF ssigma patL*]
obtain *pat* **where** *pat*: *pat* ∈ *set patL* **and** *match*: *?match-lhs* (*pat* · *σ*) **by** *auto*
have *id*: *map* (*λ pat*. *pat* · *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 ∃ *ti* ∈ *set* (*term-and-args f ts*). *?match-lhs ti* **unfolding** *id*[*symmetric*] **using** *pat match* **by** *auto*
qed
also have ... = (∀ *t*. *t* ∈ *B*(*?C*,*?D*,∅::*unit*⇀-) ⟶ (∃ *ti* ∈ *set* (*t* # *args t*). *?match-lhs ti*))
unfolding *basic-terms-def term-and-args-def* **by** *fastforce*
finally show *?thesis* **unfolding** *strong-quasi-reducible-def* **by** *blast*
qed

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

definition *quasi-reducible* :: ('*f*,'*s*)*ssig* ⇒ ('*f*,'*s*)*ssig* ⇒ ('*f*,'*v*)*term set* ⇒ *bool* **where**
quasi-reducible C D L = (∀ *t* ∈ *B*(*C*,*D*,∅::*unit*⇀'*s*). ∃ *tp* ⊴ *t*. ∃ *l* ∈ *L*. *l matches tp*)

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

lemma *arg-imp-subt*: *s* ∈ *set* (*args t*) ⟹ *t* ⊵ *s*
by (*cases t*, *auto*)

lemma *strong-quasi-reducible-imp-quasi-reducible*:
strong-quasi-reducible C D L ⟹ *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 ⋀ *l f ls τs τ*. *l* ∈ *L* ⟹ *l* = *Fun f ls* ⟹ ¬ *f* : *τs* → *τ in C*
shows *pat-complete-lhss C D L* ⟷ *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*,*unit*)*term*
assume *t*: *t* ∈ *B*(*C*,*D*,∅)

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* $\mathcal{T}(C)$ **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** $f : ss \to s$ *in* C
by (*meson Fun-hastype hastype-def fun-hastype-def*)
with *assms*[*OF l Fun*, *of ss s*] **show** *?thesis* **by** *auto*
qed
qed
qed
qed

end

10 Setup for Experiments

theory *Test-Pat-Complete*
imports
Pattern-Completeness
HOL−Library.Code-Abstract-Char
HOL−Library.Code-Target-Numeral
HOL−Library.RBT-Mapping
HOL−Library.Product-Lexorder
HOL−Library.List-Lexorder
Show.Number-Parser
begin

turn error message into runtime error

definition *pat-complete-alg* :: $((\prime f \times \prime s\ list) \times \prime s)list \Rightarrow ((\prime f \times \prime s\ list) \times \prime s)list \Rightarrow (\prime f,\prime v)term\ list \Rightarrow bool$ **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* :: $((\prime f \times \prime s\ list) \times \prime s)list \Rightarrow ((\prime f \times \prime s\ list) \times \prime s)list \Rightarrow (\prime f,\prime v)term\ list \Rightarrow bool$ **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 *rn-string* **where** *rn-string x* = "*x*" @ *show* (*x* :: *nat*)
definition *rv-string* **where** *rv-string x* = "*y*" @ *x*

lemma *renaming-string*: *renaming-funs rn-string rv-string*
using *inj-show-nat*
unfolding *renaming-funs-def*
by (*auto simp*: *inj-def rn-string-def rv-string-def*)

definition *decide-pat-complete-lhss-fidl-string* = *decide-pat-complete-lhss-fidl rn-string rv-string*

lemmas *decide-pat-complete-lhss-fidl-string* = *decide-pat-complete-lhss-fidl*[*OF* - *renaming-string*,
folded decide-pat-complete-lhss-fidl-string-def]

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-pat-complete-wrapper-fidl*:
 assumes (*case decide-pat-complete-lhss-fidl-string solver C D lhss of Inr b* ⇒ *Some b* | *Inl -* ⇒ *None*) = *Some res*
  and *finite-idl-solver solver*
 shows *pat-complete-lhss* (*map-of C*) (*map-of D*) (*set lhss*) = *res*
 using *decide-pat-complete-lhss-fidl-string*[*of solver 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*+

value *decide-pat-complete-linear-lhss int-bool even-int even-lhss-int*

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+*

value *decide-pat-complete-linear-lhss nat-bool* [((*''f''*,[*''nat''*,*''nat''*,*''nat''*]),*''bool''*)]
non-lin-lhss

value *decide-pat-complete-lhss nat-bool* [((*''f''*,[*''nat''*,*''nat''*,*''nat''*]),*''bool''*)] *non-lin-lhss*

value *decide-pat-complete-lhss nat-bool* [((*''f''*,[*''bool''*,*''bool''*,*''bool''*]),*''bool''*)] *non-lin-lhss*

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-fidl*[*of dummy-fidl-solver - - - False*])
apply *eval*
apply (*rule dummy-fidl-solver*)
apply *eval*
done

value *decide-pat-complete-lhss-fidl-string* (λ *-. True*) *nat-bool* [((*''f''*,[*''bool''*,*''bool''*,*''bool''*]),*''bool''*)]
non-lin-lhss
value *decide-pat-complete-lhss-fidl-string* (λ *-. False*) *nat-bool* [((*''f''*,[*''bool''*,*''bool''*,*''bool''*]),*''bool''*)]
non-lin-lhss

```
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↩↩data B = TT | FF↩↩''
+
     foldr (λ l s. (term-to-haskell l + STR '' = TT↩'' + 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* ''↩''
+++ *STR* ''and transitions:↩'' +++ *paren* (*STR* ''''') (*STR* ''''') (*STR* ''''') (*STR* ''↩'') *rls* +++ *STR* ''↩↩'') *n c perms*
of (*all*,*pats*) ⇒ *STR* ''decide whether language of first automaton is subset of the second automaton↩↩''
+++ *STR* ''first '' +++ *all* +++ *STR* ''↩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.