

Verifying a Decision Procedure for Pattern Completeness*

René Thiemann

University of Innsbruck, Austria

Akihisa Yamada

National Institute of Advanced Industrial Science and Technology,
Japan

June 15, 2026

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

1	Introduction	2
2	Auxiliary Algorithm for Testing Whether "set xs" is a Singleton Set	3
3	An Interface for Solvers for a Subset of Finite Integer Difference Logic	3
4	Pattern Completeness	4

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

5	A Set-Based Inference System to Decide Pattern Completeness	4
5.1	Defining Pattern Completeness	7
5.2	Definition of Algorithm – Inference Rules	10
5.3	Soundness of the inference rules	15
6	A Multiset-Based Inference System to Decide Pattern Completeness	47
6.1	Definition of the Inference Rules	47
6.2	The evaluation cannot get stuck	49
6.3	Termination	63
6.4	Partial Correctness via Refinement	87
7	A List-Based Implementation to Decide Pattern Completeness	95
7.1	Definition of Algorithm	95
7.2	Partial Correctness of the Implementation	102
7.3	Getting the result outside the locale with assumptions	167
8	Pattern-Completeness and Related Properties	323
8.1	Connecting Pattern-Completeness, Strong Quasi-Reducibility and Quasi-Reducibility	334
9	Setup for Experiments	335
9.1	FSCD paper	336
9.2	Journal Submission	342
9.3	Export Code to SML and Haskell	345

1 Introduction

This AFP entry includes the formalization of a decision procedure [4] for pattern completeness, as well as an improved version that is currently reviewed. 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  $\Rightarrow$  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  $\longleftrightarrow$  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  $\longleftrightarrow$  ( $\exists$  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  $\times$  's)  $\times$  int) list  $\times$  (('v  $\times$  's)  $\times$  'v  $\times$  's) list list
```

```
definition fidl-input :: ('v,'s)fidl-input  $\Rightarrow$  bool where
```

$fidl\text{-}input = (\lambda (bnds, diffs).$
 $distinct (map\ fst\ bnds) \wedge (\forall v\ w\ u. (v,w) \in set\ (concat\ diffs) \longrightarrow u \in \{v,w\}$
 $\longrightarrow u \in fst\ 'set\ bnds)$
 $\wedge (\forall v\ w. (v,w) \in set\ (concat\ diffs) \longrightarrow snd\ v = snd\ w)$
 $\wedge (\forall v\ w. (v,w) \in set\ (concat\ diffs) \longrightarrow v \neq w)$
 $\wedge (\forall v\ w\ b1\ b2. (v,b1) \in set\ bnds \longrightarrow (w,b2) \in set\ bnds \longrightarrow snd\ v = snd\ w$
 $\longrightarrow b1 = b2)$
 $\wedge (\forall v\ b. (v,b) \in set\ bnds \longrightarrow b \geq 0))$

definition *fidl-solvable* :: ('v,'s)fidl-input \Rightarrow bool **where**

$fidl\text{-}solvable = (\lambda (bnds, diffs). (\exists \alpha :: 'v \times 's \Rightarrow int.$
 $(\forall (v,b) \in set\ bnds. 0 \leq \alpha\ v \wedge \alpha\ v \leq b) \wedge$
 $(\forall c \in set\ diffs. \exists (v,w) \in set\ c. \alpha\ v \neq \alpha\ w)))$

definition *finite-idl-solver* **where** *finite-idl-solver solver* = (\forall input.
fidl-input input \longrightarrow *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 Pattern Completeness

Pattern-completeness is the question whether in a given program all terms of the form $f(c1, \dots, cn)$ 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 $(f(x1, \dots, xn), lhs1, \dots, lhsn)$ where the xi will represent arbitrary constructor terms.

5 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
  Sorted-Terms.Compute-Nonempty-Infinite-Sorts
begin

lemma ball-insert-un-cong:  $f y = \text{Ball } zs \ f \implies \text{Ball } (\text{insert } y \ A) \ f = \text{Ball } (zs \cup A) \ f$ 
by auto

lemma bex-insert-cong:  $f y = f z \implies \text{Bex } (\text{insert } y \ A) \ f = \text{Bex } (\text{insert } z \ A) \ f$ 
by auto

lemma not-bdd-above-natD:
  assumes  $\neg \text{bdd-above } (A :: \text{nat set})$ 
  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:  $xs = ys \iff \text{length } xs = \text{length } ys \wedge (\forall i < \text{length } ys. xs ! i = ys ! i)$ 
using nth-equalityI by metis

lemma subt-size:  $p \in \text{poss } t \implies \text{size } (t \mid\!-\! p) \leq \text{size } t$ 
proof (induct p arbitrary: t)
  case (Cons i p t)
  thus ?case
  proof (cases t)
    case (Fun f ss)
    from Cons Fun have  $i < \text{length } ss$  and  $\text{sub: } t \mid\!-\! (i \# p) = (ss ! i) \mid\!-\! p$ 
    and  $p \in \text{poss } (ss ! i)$  by auto
    with Cons(1)[OF this(3)]
    have  $\text{size } (t \mid\!-\! (i \# p)) \leq \text{size } (ss ! i)$  by auto
    also have  $\dots \leq \text{size } t$  using i unfolding Fun by (simp add: termination-simp)
    finally show ?thesis .
  qed auto
qed auto

lemma removeAll-remdups:  $\text{removeAll } x \ (\text{remdups } ys) = \text{remdups } (\text{removeAll } x \ ys)$ 
by (simp add: remdups-filter removeAll-filter-not-eq)

lemma removeAll-eq-Nil-iff:  $\text{removeAll } x \ ys = [] \iff (\forall y \in \text{set } ys. y = x)$ 
by (induction ys, auto)

lemma concat-removeAll-Nil:  $\text{concat } (\text{removeAll } [] \ xss) = \text{concat } xss$ 
by (induction xss, auto)

```

lemma *removeAll-eq-imp-concat-eq*:
assumes *removeAll [] xs = removeAll [] xss'*
shows *concat xs = 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*: $\exists_{\leq 1} a. \text{False}$ **by** (*auto intro!: Uniq-I*)

abbreviation *UNIQ A* $\equiv \exists_{\leq 1} a. a \in A$

lemma *Uniq-eq-the-elem*:
assumes *UNIQ A* **and** $a \in A$ **shows** $a = \text{the-elem } A$
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** $\text{UNIQ } A \longleftrightarrow \text{UNIQ } B$
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*: $\text{UNIQ } A \implies \text{UNIQ } (f \text{' } A)$
by (*smt (verit) Uniq-I image-iff the1-equality'*)

lemma *successively-eq-iff-Uniq*: $\text{successively } (=) \text{ } xs \longleftrightarrow \text{UNIQ } (\text{set } xs)$ (**is** $?l \longleftrightarrow ?r$)

proof
show $?l \implies ?r$
apply (*induction xs rule: induct-list012*)
by (*auto intro: Uniq-I*)
show $?r \implies ?l$
proof (*induction xs*)
case *Nil*
then show $?case$ **by** *simp*
next
case $xs: (\text{Cons } x \text{ } xs)$
show $?case$
proof (*cases xs*)
case *Nil*
then show $?thesis$ **by** *simp*
next
case $xs: (\text{Cons } y \text{ } ys)$
have $\text{successively } (=) \text{ } xs$

```

    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

```

5.1 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 $tvars :: nat \times 's \rightarrow 's \ (\mathcal{V})$ **where** $\mathcal{V} \equiv \text{sort-annotated}$

type-synonym $(f, 'v, 's)\text{match-atom} = (f, nat \times 's)\text{term} \times (f, 'v)\text{term}$

type-synonym $(f, 'v, 's)\text{match-problem-set} = (f, 'v, 's)\ \text{match-atom set}$

type-synonym $(f, 'v, 's)\text{pat-problem-set} = (f, 'v, 's)\text{match-problem-set set}$

type-synonym $(f, 'v, 's)\text{pats-problem-set} = (f, 'v, 's)\text{pat-problem-set set}$

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

definition $tvars\text{-match} :: (f, 'v, 's)\text{match-problem-set} \Rightarrow (nat \times 's)\ \text{set}$ **where**
 $tvars\text{-match}\ mp = (\bigcup (t, l) \in mp. vars\ t)$

definition $tvars\text{-pat} :: (f, 'v, 's)\text{pat-problem-set} \Rightarrow (nat \times 's)\ \text{set}$ **where**
 $tvars\text{-pat}\ pp = (\bigcup mp \in pp. tvars\text{-match}\ mp)$

definition $tvars\text{-pats} :: (f, 'v, 's)\text{pats-problem-set} \Rightarrow (nat \times 's)\ \text{set}$ **where**
 $tvars\text{-pats}\ P = (\bigcup pp \in P. tvars\text{-pat}\ pp)$

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

A definition of pattern completeness for pattern problems.

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

lemma $match\text{-complete-wrt-cong}$:

assumes $s: \bigwedge x. x \in tvars\text{-match}\ mp \implies \sigma\ x = \sigma'\ x$

and $mp: mp = mp'$
shows $match\text{-}complete\text{-}wrt\ \sigma\ mp = match\text{-}complete\text{-}wrt\ \sigma'\ mp'$
apply (*unfold match-complete-wrt-def Ball-Pair-conv mp[symmetric]*)
apply (*intro ex-cong1 all-cong1 imp-cong[OF refl]*)
proof –
fix $\mu\ t\ l$ **assume** $(t,l) \in mp$
with s **have** $\forall x \in vars\ t.\ \sigma\ x = \sigma'\ x$ **by** (*auto simp: tvars-match-def*)
from *subst-same-vars[OF this]* **show** $t \cdot \sigma = l \cdot \mu \longleftrightarrow t \cdot \sigma' = l \cdot \mu$ **by** *simp*
qed

lemma *match-complete-wrt-imp-o*:
assumes $match\text{-}complete\text{-}wrt\ \sigma\ mp$ **shows** $match\text{-}complete\text{-}wrt\ (\sigma \circ_s \tau)\ mp$
proof (*unfold match-complete-wrt-def*)
from *assms[unfolded match-complete-wrt-def]* **obtain** μ **where** $eq: \forall (t,l) \in mp.$
 $t \cdot \sigma = l \cdot \mu$
by *auto*
{ **fix** $t\ l$
assume $tl: (t,l) \in mp$
with eq **have** $t \cdot (\sigma \circ_s \tau) = l \cdot (\mu \circ_s \tau)$ **by** *auto*
}
then show $\exists \mu'. \forall (t,l) \in mp. t \cdot (\sigma \circ_s \tau) = l \cdot \mu'$ **by** *blast*
qed

lemma *match-complete-wrt-o-imp*:
assumes $s: \sigma :_s \mathcal{V} \mid' tvars\text{-}match\ mp \rightarrow \mathcal{T}(C, \emptyset)$ **and** $m: match\text{-}complete\text{-}wrt\ (\sigma$
 $\circ_s \tau)\ mp$
shows $match\text{-}complete\text{-}wrt\ \sigma\ mp$
proof (*unfold match-complete-wrt-def*)
from $m[unfolded\ match\text{-}complete\text{-}wrt\text{-}def]$ **obtain** μ **where** $eq: \forall (t,l) \in mp. t \cdot \sigma \cdot \tau$
 $= l \cdot \mu$
by *auto*
have $\forall x \in tvars\text{-}match\ mp. \sigma\ x : snd\ x\ in\ \mathcal{T}(C, \emptyset)$
by (*auto intro!: sorted-mapD[OF s] simp: hastype-restrict*)
then have $g: x \in tvars\text{-}match\ mp \implies ground\ (\sigma\ x)$ **for** x
by (*auto simp: Term-empty-imp-ground*)
{ **fix** $t\ l$
assume $tl: (t,l) \in mp$
then have $ground\ (t \cdot \sigma)$ **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* $:: (f, 's)\ ssig \Rightarrow (f, 'v, 's)\ pat\text{-}problem\text{-}set \Rightarrow bool$ **where**
 $pat\text{-}complete\ C\ pp \longleftrightarrow (\forall \sigma :_s \mathcal{V} \mid' tvars\text{-}pat\ pp \rightarrow \mathcal{T}(C). \exists mp \in pp. match\text{-}complete\text{-}wrt$

σ mp)

lemma *pat-completeD*:

assumes pp : *pat-complete* C pp

and s : $\sigma :_s \mathcal{V} \mid 'tvars$ -*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* $:_s \mathcal{V} \mid 'tvars$ -*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 $\sigma :_s \mathcal{V} \mid 'tvars$ -*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 : $\forall \sigma :_s \mathcal{V} \mid 'tvars$ -*pat* $pp \rightarrow \mathcal{T}(C, \emptyset :: 'v \rightarrow 's)$. $\exists mp \in pp$. *match-complete-wrt* σ mp

shows *pat-complete* C pp

proof (*unfold pat-complete-def, safe*)

fix σ **assume** s : $\sigma :_s \mathcal{V} \mid 'tvars$ -*pat* $pp \rightarrow \mathcal{T}(C)$

then have $\sigma \circ_s$ *undefined* $:_s \mathcal{V} \mid 'tvars$ -*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 $\sigma :_s \mathcal{V} \mid 'tvars$ -*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* $\Rightarrow ('f, 'v, 's)$ *pats-problem-set* $\Rightarrow bool$ **where**
pats-complete C $P \equiv \forall pp \in P$. *pat-complete* C pp

definition *finite-constr-form-mp* :: ('f,'s) ssig \Rightarrow ('f,'v,'s)match-problem-set \Rightarrow bool **where**

finite-constr-form-mp C mp = (\forall t l. (t,l) \in mp \longrightarrow is-Var l \wedge (\exists ι . finite-sort C ι \wedge t : ι in $\mathcal{T}(C,\mathcal{V})$))

definition *finite-constr-form-pat* :: ('f,'s) ssig \Rightarrow ('f,'v,'s)pat-problem-set \Rightarrow bool **where**

finite-constr-form-pat C p = (\forall mp \in p. *finite-constr-form-mp* C mp)

5.2 Definition of Algorithm – Inference Rules

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

definition τc :: nat \Rightarrow nat \times 's \Rightarrow 'f \times 's list \Rightarrow ('f,nat \times 's)subst **where**

τc n x = (λ (f, ss). subst x (Fun f (map Var (indexed-from n ss))))

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

fun *conflicts* :: ('f,'v \times 's)term \Rightarrow ('f,'v \times 's)term \Rightarrow ('v \times '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 \equiv *conflicts* s t \neq None \wedge x \in set (the (*conflicts* s t))

abbreviation *Conflict-Clash* s t \equiv *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 (λ 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) λ [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 (λ s ss t ts)

hence IH: rel-option (λ x y. \bigcup (set ' set x) = \bigcup (set ' set y)) (those (map2 c ss ts)) (those (map2 c ts ss)) **by** auto

```

from 2 have st: rel-option ( $\lambda xs\ ys. set\ xs = set\ ys$ ) (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  $\implies$ 
   $\exists p. p \in poss\ s \wedge p \in poss\ t \wedge$ 
  (is-Fun (s |-p)  $\wedge$  is-Fun (t |-p)  $\wedge$  root (s |-p)  $\neq$  root (t |-p)  $\vee$ 
  ( $\exists x\ y. s\ |-p = Var\ x \wedge t\ |-p = Var\ y \wedge snd\ x \neq snd\ y$ ))
  (is ?B1  $\implies$  ?B2)
and Conflict-Var s t x  $\implies$ 
   $\exists p. p \in poss\ s \wedge p \in poss\ t \wedge s\ |-p \neq t\ |-p \wedge$ 
  (s |-p = Var x  $\vee$  t |-p = Var x)
  (is ?C1 x  $\implies$  ?C2 x)
and s  $\neq$  t  $\implies \exists x. Conflict-Clash\ s\ t \vee Conflict-Var\ s\ t\ x$ 
and Conflict-Var s t x  $\implies x \in vars\ s \cup vars\ t$ 
and conflicts s t = Some []  $\longleftrightarrow s = t$  (is ?A)

```

proof –

```

let ?B = ?B1  $\longrightarrow$  ?B2
let ?C =  $\lambda x. ?C1\ x \longrightarrow ?C2\ x$ 
{
  fix x
  have (conflicts s t = Some []  $\longrightarrow s = t$ )  $\wedge$  ?B  $\wedge$  ?C x
  proof (induction s arbitrary: t)
    case (Var y t)
    thus ?case by (cases t, 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  $ss ! i \in \text{set } ss$  by auto
    from  $\text{Fun.IH}[OF \text{ this, of } ts ! i, \text{ unfolded } res]$  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  $main = \text{this}$ 
from  $main$  show  $B: ?B1 \implies ?B2$  and  $C: ?C1 x \implies ?C2 x$  by blast+
show  $?A$ 
proof
  assume  $s = t$ 
  with  $B$  have  $\text{conflicts } s t \neq \text{None}$  by force
  then obtain  $xs$  where  $res: \text{conflicts } s t = \text{Some } xs$  by auto
  show  $\text{conflicts } s t = \text{Some } []$ 
  proof (cases  $xs$ )
    case Nil
    thus  $?thesis$  using  $res$  by auto
  next
    case (Cons  $x xs$ )
    with  $main[of x] res \langle s = t \rangle$  show  $?thesis$  by auto
  qed
qed (insert  $main, blast$ )
{
  assume  $diff: s \neq t$ 
  show  $\exists x. \text{Conflict-Clash } s t \vee \text{Conflict-Var } s t x$ 
  proof (cases  $\text{conflicts } s t$ )
    case (Some  $xs$ )
    with  $\langle ?A \rangle diff$  obtain  $x$  where  $x \in \text{set } xs$  by (cases  $xs, auto$ )
    thus  $?thesis$  unfolding  $\text{Some}$ 
    apply auto
    by (metis surj-pair)
  qed auto
}
assume  $\text{Conflict-Var } s t x$ 
with  $C$  obtain  $p$  where  $p \in \text{poss } s p \in \text{poss } t (s \mid - p = \text{Var } x \vee t \mid - p = \text{Var } x)$ 
  by blast
  thus  $x \in \text{vars } s \cup \text{vars } t$ 
  by (metis UnCI subt-at-imp-supteq' subteq-Var-imp-in-vars-term)
qed

declare  $\text{conflicts.simps}[simp del]$ 

lemma  $\text{conflicts-refl}[simp]: \text{conflicts } t t = \text{Some } []$ 

```

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 (journal: section 4); if improved = True, the co-NP algorithm of the journal version is used (section 5).
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* ⇒ insert (Fun *f ts*, Fun *f ls*) *mp* →_{*s*} set (zip *ts ls*) ∪ *mp*

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

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

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

Note that in $pp\text{-inf-}var\text{-conflict}$ the conflicts have to be simultaneously occurring. If just some matching problem has such a conflict, then this cannot be deleted immediately!

Example-program: $f(x,x) = \dots, f(s(x),y) = \dots, f(x,s(y)) = \dots$ 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 $inf\text{-}var\text{-conflict}$ in $(x1,x), (x2,x)$ then it is no longer complete.

inductive $P\text{-step-set} :: ('f, 'v, 's)pat\text{-problem-set} \Rightarrow ('f, 'v, 's)pat\text{-problem-set} \Rightarrow bool$
(infix $\langle \Rightarrow_s \rangle$ **50)** **where**
 $P\text{-fail}: insert\ \{\}\ P \Rightarrow_s bottom$
 $| P\text{-simp}: pp \Rightarrow_s P' \implies insert\ pp\ P \Rightarrow_s P' \cup P$

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

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

lemma *tvars-match-subst*: $tvars\text{-match } (subst\text{-match-problem-set } \sigma mp) = (\bigcup (t,l) \in mp. vars (t.\sigma))$
by (*auto simp: tvars-match-def subst-match-problem-set-def subst-left-def*)

lemma *wf-match-subst*:

assumes $s: \sigma :_s \mathcal{V} \mid ' tvars\text{-match } mp \rightarrow \mathcal{T}(C', \{x : \iota \text{ in } \mathcal{V}. \iota \in S\})$

shows $wf\text{-match } (subst\text{-match-problem-set } \sigma mp)$

apply (*unfold wf-match-iff tvars-match-subst*)

proof (*safe*)

fix $t \ l \ x \ \iota$ **assume** $tl: (t,l) \in mp$ **and** $xt: (x,\iota) \in vars (t.\sigma)$

from xt **obtain** $y \ \kappa$ **where** $y: (y,\kappa) \in vars t$ **and** $xy: (x,\iota) \in vars (\sigma (y,\kappa))$ **by**
(*auto simp: vars-term-subst*)

from $tl \ y$ **have** $(y,\kappa) : \kappa \text{ in } \mathcal{V} \mid ' tvars\text{-match } mp$ **by** (*auto simp: hastype-restrict tvars-match-def*)

from *sorted-mapD*[*OF s this*]

have $\sigma (y,\kappa) : \kappa \text{ in } \mathcal{T}(C', \{x : \iota \text{ in } \mathcal{V}. \iota \in S\})$.

from *hastype-in-Term-imp-vars*[*OF this xy*]

have $(x,\iota) : \iota \text{ in } \{x : \iota \text{ in } \mathcal{V}. \iota \in S\}$ **by** (*auto elim!: in-dom-hastypeE*)

then show $\iota \in S$ **by** *auto*

qed

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

lemma *wf-pat-subst*:

assumes $s: \sigma :_s \mathcal{V} \mid ' tvars\text{-pat } pp \rightarrow \mathcal{T}(C', \{x : \iota \text{ in } \mathcal{V}. \iota \in S\})$

shows $wf\text{-pat } (subst\text{-pat-problem-set } \sigma pp)$

apply (*unfold wf-pat-def subst-pat-problem-set-def*)

proof *safe*

fix mp **assume** $mp: mp \in pp$

show $wf\text{-match } (subst\text{-match-problem-set } \sigma mp)$

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

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

The reduction of match problems preserves completeness.

lemma *mp-step-pcorrect*: $mp \rightarrow_s mp' \Longrightarrow match\text{-complete-wrt } \sigma mp = match\text{-complete-wrt } \sigma mp'$

```

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  $\mu$  where eq:  $\bigwedge ti li. (ti, li) \in mp \implies ti \cdot \sigma = li \cdot \mu$  by auto
    let ? $\mu = \mu(x := t \cdot \sigma)$ 
    have  $(ti, li) \in mp \implies ti \cdot \sigma = li \cdot ?\mu$  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 - ? $\mu$ ],
  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  $\sigma$  (?mpi  $\cup$  mp')
    from this[unfolded match-complete-wrt-def] obtain  $\mu$ 
      where match:  $\bigwedge t l. (t, l) \in ?mpi \implies t \cdot \sigma = l \cdot \mu$ 
      and match':  $\bigwedge t l. (t, l) \in mp' \implies t \cdot \sigma = l \cdot \mu$  by force
    let ? $\mu = \mu(y := Fun f (map \mu ys))$ 
    show match-complete-wrt  $\sigma$  (mp  $\cup$  mp') unfolding match-complete-wrt-def
    proof (intro exI[of - ? $\mu$ ] 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  $(ti, yi) \in set (zip ts ys)$ 
        hence  $(ti, Var yi) \in set (zip (args t) (map Var ys))$ 
          using t lts  $\langle$ length ys = n $\rangle$  by (force simp: set-conv-nth)
        hence  $(ti, Var yi) \in ?mpi$  using tl by blast
        from match[OF this] have  $\mu yi = ti \cdot \sigma$  by simp
      } note yi = this
  
```

```

  show  $t \cdot \sigma = l \cdot ?\mu$  unfolding  $l t$  using  $yi\ lts\ \langle length\ ys = n \rangle$ 
    by (force intro!: nth-equalityI simp: set-zip)
qed
next
assume match-complete-wrt  $\sigma$  ( $mp \cup mp'$ )
from this[unfolded match-complete-wrt-def]
obtain  $\mu$  where match:  $\bigwedge t\ l. (t,l) \in mp \implies t \cdot \sigma = l \cdot \mu$ 
  and match':  $\bigwedge t\ l. (t,l) \in mp' \implies t \cdot \sigma = l \cdot \mu$  by force
define  $\mu'$  where  $\mu' = (\lambda x. case\ map-of\ (zip\ ys\ (args\ (\mu\ y)))\ x\ of$ 
  None  $\Rightarrow \mu\ x \mid Some\ Ti \Rightarrow Ti)$ 
show match-complete-wrt  $\sigma$  ( $?mpi \cup mp'$ )
  unfolding match-complete-wrt-def
proof (intro exI[of -  $\mu'$ ] ballI, elim UnE; clarify)
  fix  $t\ l$ 
  assume  $tl: (t,l) \in mp'$ 
  from *(3)  $tl$  have vars:  $vars\ l \cap set\ ys = \{\}$  by force
  hence map-of (zip ys (args ( $\mu\ y$ )))  $x = None$  if  $x \in vars\ l$  for  $x$ 
  using that by (meson disjoint-iff map-of-SomeD option.exhaust set-zip-leftD)
  with match'[OF  $tl$ ]
  show  $t \cdot \sigma = l \cdot \mu'$  by (auto intro!: term-subst-eq simp:  $\mu'$ -def)
next
fix  $t\ l\ ti$  and  $v yi :: ('f,-)term$ 
assume  $tl: (t,l) \in mp$ 
  and  $i: (ti,v yi) \in set\ (zip\ (args\ t)\ (map\ Var\ ys))$ 
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  $yi: v yi = Var\ (ys$ 
!  $i)$ 
  unfolding set-zip using  $\langle length\ ys = n \rangle\ t$  by auto
from match[OF  $tl$ ] have  $mu-y: \mu\ y = Fun\ f\ ts \cdot \sigma$  unfolding  $l\ t$  by auto
have  $yi: v yi \cdot \mu' = args\ (\mu\ y)\ !\ i$  unfolding  $\mu'$ -def  $yi$ 
  using  $i\ lts\ \langle length\ ys = n \rangle\ *(3)\ mu-y$ 
  by (force split: option.splits simp: set-zip distinct-conv-nth)
also have  $\dots = ti \cdot \sigma$  unfolding  $mu-y\ ti$  using  $i\ lts$  by auto
finally show  $ti \cdot \sigma = v yi \cdot \mu' ..$ 
qed
qed
qed auto

lemma mp-fail-pcorrect1:
  assumes mp-fail  $mp\ \sigma :_s$  sort-annotated  $|^t\ tvar$ s-match  $mp \rightarrow \mathcal{T}(C,X)$ 
  shows  $\neg$  match-complete-wrt  $\sigma\ mp$ 
  using assms
proof (induct mp rule: mp-fail.induct)
  case *: (mp-clash  $f\ ts\ g\ ls\ mp)$ 
  {
    assume  $length\ ts \neq length\ ls$ 
    hence (map ( $\lambda t. t \cdot \mu$ )  $ls = map\ (\lambda t. t \cdot \sigma)\ ts = False$  for  $\sigma :: ('f,nat \times$ 
 $'s,'a)gsubst$  and  $\mu$ 

```

```

    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  $\mathcal{T}(C, X)$  σ z : snd z in  $\mathcal{T}(C, X)$ 
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)} ∪ mp)
  from this[unfolded match-complete-wrt-def]
  have eq: s · σ = t · σ by auto
  define V where V = tvars-match ({(s, Var x), (t, Var x)} ∪ mp)
  from *(2) have σ: σ :s  $\mathcal{V}$  |' V →  $\mathcal{T}(C, X)$  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  $\sigma$  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  $\sigma$  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  $\sigma t$  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  $\sigma$  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  $\sigma s$ )
  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  $\sigma$  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  $\cdot$   $\sigma$  :  $\sigma s$  in  $\mathcal{T}(C, X)$   $\mathcal{T}(C, \mathcal{V})$  s = None
by auto
thus False
proof (induct s arbitrary:  $\sigma s$ )
  case (Fun f ss  $\tau$ )
  hence mem: Fun f (map ( $\lambda s$ . s  $\cdot$   $\sigma$ ) ss) :  $\tau$  in  $\mathcal{T}(C, X)$  by auto
  from this[unfolded Fun-hastype]
  obtain  $\tau s$  where f: f :  $\tau s \rightarrow \tau$  in C and args: map ( $\lambda s$ . s  $\cdot$   $\sigma$ ) ss :l  $\tau s$  in
 $\mathcal{T}(C, X)$  by auto
  {
    fix s

```

```

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

```

lemma *mp-fail-pcorrect:*

```

assumes  $f : \text{mp-fail } mp$  and  $s : \sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$  and  $wf : wf\text{-match}$ 
 $mp$ 
shows  $\neg \text{match-complete-wrt } \sigma\ mp$ 
apply (rule mp-fail-pcorrect1[OF f])
apply (rule sorted-map-cmono[OF s])
using  $wf$  by (auto intro!: subsetI simp: hastype-restrict wf-match-iff)

```

abbreviation SS **where** $SS \equiv (\text{UNIV} :: \text{nat set}) \times S$

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* k +
assumes *not-empty-sort*: $\bigwedge s. s \in S \implies \neg \text{empty-sort } C s$
and *C-sub-S*: $\bigwedge f ss s. f : ss \rightarrow s \text{ in } C \implies \text{insert } s (\text{set } ss) \subseteq S$
and m : $\bigwedge f ss s. f : ss \rightarrow s \text{ in } C \implies \text{length } ss \leq m$
and *finite-C*: *finite* (*dom* C)
and *inf-sort*: $\bigwedge s. s \in S \implies \text{inf-sort } s \longleftrightarrow \neg \text{finite-sort } C s$
and Cl : $\bigwedge s. \text{set } (Cl s) = \{(f, ss). f : ss \rightarrow s \text{ in } C\}$
and *Cl-len*: $\bigwedge \sigma. \text{Ball } (\text{length } 'snd \text{ ' set } (Cl \sigma)) (\lambda a. a \leq m)$
and *cd*: $\bigwedge s. s \in S \implies \text{cd-sort } s = \min k (\text{card-of-sort } C s)$
and $k1$: $k > 1$

begin

lemma *sorts-non-empty*: $s \in S \implies \exists t. t : s \text{ in } \mathcal{T}(C, \emptyset)$

apply (*drule not-empty-sort*)

by (*auto elim: not-empty-sortE*)

lemma *inf-sort-not-bdd*: $s \in S \implies \neg \text{bdd-above } (\text{size } ' \{t . t : s \text{ in } \mathcal{T}(C, \emptyset)\}) \longleftrightarrow$
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*: $f : ss \rightarrow s \text{ in } C \implies i < \text{length } ss \implies ss!i \in S$

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*: $mp \rightarrow_s mp' \implies \text{wf-match } mp \implies \text{wf-match } mp'$

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  $\Rightarrow_s$  P'  $\Longrightarrow$  wf-pat pp  $\Longrightarrow$  pp'  $\in$  P'  $\Longrightarrow$  wf-pat pp'
  unfolding wf-pat-def wf-pats-def
proof (induct pp P' rule: pp-step.induct)
  case (pp-simp-mp mp mp' pp)
  then show ?case using mp-step-wf[of mp mp'] by auto
next
  case *: (pp-instantiate n pp x)
  let ?s = snd x
  from * have sS: ?s  $\in$  S and p: wf-pat pp unfolding wf-pat-def wf-match-def
  tvars-pat-def by auto
  {
    fix  $\tau$ 
    assume tau:  $\tau \in \tau s n x$ 
    from tau[unfolded  $\tau s$ -def  $\tau c$ -def, simplified]
    obtain f sorts where f: f : sorts  $\rightarrow$  snd x in C
    and  $\tau$ :  $\tau = \text{subst } x (\text{Fun } f (\text{map } \text{Var } (\text{indexed-from } n \text{ sorts})))$  by auto
    let ?i = length sorts
    let ?xs = indexed-from n sorts
    from C-sub-S[OF f] have sS: ?s  $\in$  S and xs: snd ' set ?xs  $\subseteq$  S
    by (auto simp add: nth-indexed-from-eq set-conv-nth set-zip)
    {
      fix mp y
      assume mp: mp  $\in$  pp and y  $\in$  tvars-match (subst-left  $\tau$  ' mp)
      then obtain s t where y: y  $\in$  vars (s  $\cdot$   $\tau$ ) and st: (s,t)  $\in$  mp
      unfolding tvars-match-def subst-left-def by auto
      from y have y  $\in$  vars s  $\cup$  set ?xs unfolding vars-term-subst  $\tau$ 
      by (auto simp: subst-def split: if-splits)
      hence snd y  $\in$  snd ' vars s  $\cup$  snd ' set ?xs by auto
      also have ...  $\subseteq$  snd ' vars s  $\cup$  S using xs by auto
      also have ...  $\subseteq$  S using p mp st
      unfolding wf-pat-def wf-match-def tvars-match-def by force
      finally have snd y  $\in$  S .
    }
    hence wf-pat (subst-pat-problem-set  $\tau$  pp)
    unfolding wf-pat-def wf-match-def subst-defs-set by auto
  }
  with * show ?case unfolding wf-pat-def by auto
qed auto

theorem P-step-set-wf: P  $\Rightarrow_s$  P'  $\Longrightarrow$  wf-pats P  $\Longrightarrow$  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

```

qed *auto*

Soundness requires some preparations

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

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

lemma *wf-pat-complete-iff:*

assumes *wf-pat pp*
shows *pat-complete C pp* $\longleftrightarrow (\forall \sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C). \exists mp \in pp. \text{match-complete-wrt } \sigma \text{ mp})$
(is ?l \longleftrightarrow **?r)**

proof

assume *l: ?l*

show *?r*

proof (*intro allI impI*)

fix $\sigma :: \text{nat} \times 's \Rightarrow -$

assume $s: \sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$

have $\sigma :_s \mathcal{V} \mid ' \text{tvars-pat pp} \rightarrow \mathcal{T}(C)$

apply (*rule sorted-map-cmono[OF s]*)

using *assms* **by** (*auto intro!: subsetI simp: hastype-restrict wf-pat-iff*)

from *pat-completeD[OF l this]* **show** $\exists mp \in pp. \text{match-complete-wrt } \sigma \text{ mp}.$

qed

next

assume *r: ?r*

show *?l*

proof (*unfold pat-complete-def, safe*)

fix σ **assume** $s: \sigma :_s \mathcal{V} \mid ' \text{tvars-pat pp} \rightarrow \mathcal{T}(C)$

define σ' **where** $\sigma' x \equiv \text{if } x \in \text{tvars-pat pp} \text{ then } \sigma x \text{ else } \sigma g x$ **for** x

have $\sigma' :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$

by (*auto intro!: sorted-mapI sorted-mapD[OF s] sorted-mapD[OF \sigma g] simp: \sigma'-def hastype-restrict*)

from *r[rule-format, OF this]*

obtain mp **where** $mp: mp \in pp$ **and** $m: \text{match-complete-wrt } \sigma' \text{ mp}$ **by** *auto*

have [*simp*]: $x \in \text{tvars-match } mp \implies \sigma x = \sigma' x$ **for** x **using** mp **by** (*auto simp: \sigma'-def tvars-pat-def*)

from m **have** *match-complete-wrt* $\sigma \text{ mp}$ **by** (*simp cong: match-complete-wrt-cong*)

with mp **show** $Bex \text{ pp} (\text{match-complete-wrt } \sigma)$ **by** *auto*

qed

qed

lemma *wf-pats-complete-iff:*

assumes *wf: wf-pats P*

shows *pats-complete C P* \longleftrightarrow

$(\forall \sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C). \forall pp \in P. \exists mp \in pp. \text{match-complete-wrt } \sigma \text{ mp})$

```

    (is ?l  $\longleftrightarrow$  ?r)
proof safe
  fix  $\sigma$  pp assume  $s: \sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$  and  $pp: pp \in P$ 
  have  $s2: \sigma :_s \mathcal{V} \mid ' \text{tvars-pats } P \rightarrow \mathcal{T}(C)$ 
    apply (rule sorted-map-cmono[OF s])
    using wf
  by (auto intro!: subsetI 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  $\exists mp \in pp. \text{match-complete-wrt } \sigma \text{ } mp.$ 
next
  fix pp assume  $pp: pp \in P$ 
  assume  $r[\text{rule-format}]: ?r$ 
  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  $r[\text{OF} - pp]$  by auto
qed

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

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

lemma  $\sigma g': \sigma g' :_s \mathcal{V} \mid ' SS \rightarrow \mathcal{T}(C)$ 
  using sorts-non-empty[where ?'a = unit, THEN someI-ex]
  by (auto intro!: sorted-mapI simp:  $\sigma g'$ -def restrict-map-def split: if-splits)

lemma typed-imp-S:  $t : \iota \text{ in } \mathcal{T}(C, \mathcal{V} \mid ' SS) \implies \iota \in S$ 
proof (induct rule: hastype-in-Term-induct)
  case (Var v  $\sigma$ )
  then show ?case by (auto simp: hastype-def restrict-map-def split: if-splits)
next
  case (Fun f ss  $\sigma s \tau$ )
  from C-sub-S[OF Fun(1)] show ?case by auto
qed

```

lemma *typed-S-eq*: **assumes** $t : \tau$ in $\mathcal{T}(C, \mathcal{V} \mid 'SS)$
and $t' : \iota$ in $\mathcal{T}(C, \mathcal{V})$
shows $\tau = \iota$
proof –
have $\text{dom } (\mathcal{V} \mid 'SS) \subseteq \text{dom } \mathcal{V}$ **by** *auto*
with t **have** $t : \tau$ in $\mathcal{T}(C, \mathcal{V})$
by (*meson hastype-in-Term-mono-right restrict-submap*)
with t' **show** $\tau = \iota$ **by** (*simp add: hastype-def*)
qed

lemma *finite-arg-sort*:
assumes *finite-sort* $C \ \iota$
and $f : \sigma s \rightarrow \iota$ in C
and $\sigma : \sigma \in \text{set } \sigma s$
shows *finite-sort* $C \ \sigma$
proof –
from $C\text{-sub-}S[OF \ f]$ **have** $\text{sub} : \text{set } \sigma s \subseteq S$ **by** *auto*
define t **where** $t \ s = (\text{SOME } t. t : s \text{ in } \mathcal{T}(C))$ **for** s
have $t : s \in \text{set } \sigma s \implies t \ s : s \text{ in } \mathcal{T}(C)$ **for** s
using *someI-ex[OF sorts-non-empty[of s]] sub*
unfolding $t\text{-def}$ **by** *auto*
from σ **obtain** i **where** $\sigma : \sigma = \sigma s ! i$ **and** $i : i < \text{length } \sigma s$ **by** (*auto simp: set-conv-nth*)
define trm **where** $\text{trm } u = \text{Fun } f \ (\text{map } (\lambda j. \text{if } j = i \text{ then } u \text{ else } t \ (\sigma s ! j)))$
 $[0..<\text{length } \sigma s]$ **for** u
{
fix u
assume $u : u : \sigma$ in $\mathcal{T}(C)$
have $\text{trm } u : \iota$ in $\mathcal{T}(C)$
unfolding trm-def
apply (*intro Fun-hastypeI[OF f] list-all2-all-nthI*)
apply *force*
subgoal for j **using** $t \ u$ **by** (*auto simp: set-conv-nth* σ)
done
}
hence $\text{trm } \{u. u : \sigma \text{ in } \mathcal{T}(C)\} \subseteq \{u. u : \iota \text{ in } \mathcal{T}(C)\}$ **by** *auto*
with *assms(1)[unfolded finite-sort-def]*
have *finite* $(\text{trm } \{u. u : \sigma \text{ in } \mathcal{T}(C)\})$ **by** (*metis finite-subset*)
moreover **have** $\text{args } (\text{trm } u) ! i = u$ **for** u **unfolding** trm-def **using** i **by** *auto*
hence *inj-on* $\text{trm } A$ **for** A **unfolding** *inj-on-def* **by** *metis*
ultimately **have** *finite* $\{u. u : \sigma \text{ in } \mathcal{T}(C)\}$ **by** (*rule finite-imageD*)
thus *?thesis* **unfolding** finite-sort-def **by** *auto*
qed

lemma *finite-arg-sorts*:
assumes *finite-sort* $C \ \iota$
and $\text{Fun } f \ ts : \iota$ in $\mathcal{T}(C, V)$
and $t \in \text{set } ts$
shows $\exists \iota. t : \iota$ in $\mathcal{T}(C, V) \wedge \text{finite-sort } C \ \iota$

```

proof –
  from assms(2)[unfolded Fun-hastype] obtain  $\sigma s$ 
    where  $f: f : \sigma s \rightarrow \iota$  in  $C$  and  $ts: ts :_{\iota} \sigma s$  in  $\mathcal{T}(C, V)$  by auto
  from assms(3) obtain  $i$  where  $i: i < \text{length } ts$  and  $t: t = ts ! i$ 
    by (auto simp: set-conv-nth)
  from  $ts$ [unfolded list-all2-conv-all-nth]  $i$ 
  have  $\text{typed}: ts ! i : \sigma s ! i$  in  $\mathcal{T}(C, V)$ 
    and  $\text{len}: \text{length } \sigma s = \text{length } ts$  by auto
  hence  $\text{sig}: \sigma s ! i \in \text{set } \sigma s$  using  $i$  by auto
  show ?thesis unfolding t
  proof (intro exI conjI, rule typed)
    show finite-sort C (σ s ! i) using finite-arg-sort[OF assms(1) f sig] .
  qed
qed

```

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

lemma *pp-step-pcorrect*:

$pp \Rightarrow_s P' \Longrightarrow \text{wf-pat } pp \Longrightarrow \text{pat-complete } C \text{ } pp = \text{pats-complete } C \text{ } P'$

proof (*induct pp P' 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 $\text{wf} = \langle \text{wf-pat } (pp \cup pp') \rangle$ **and** $\text{fin} = \langle \text{finite } pp \rangle$

hence *wf-pat pp* **and** *wfpp'*: *wf-pat pp'* **by** (*auto simp: wf-pat-def*)

with *wf* **have** *easy: pat-complete C pp' \Longrightarrow pat-complete C (pp \cup pp')*

by (*auto simp: wf-pat-complete-iff*)

{

assume $pp: \text{pat-complete } C \text{ } (pp \cup pp')$

have $\text{pat-complete } C \text{ } pp'$ **unfolding** *wf-pat-complete-iff*[*OF wfpp'*]

proof (*intro allI impI*)

fix δ

assume $\delta: \delta :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$

define $\text{conv} :: ('f, \text{unit}) \text{ term} \Rightarrow ('f, \text{nat} \times 's) \text{ term}$ **where** $\text{conv } t = t \cdot$

undefined **for** t

define $\text{conv}' :: ('f, \text{nat} \times 's) \text{ term} \Rightarrow ('f, \text{unit}) \text{ term}$ **where** $\text{conv}' t = t \cdot$

undefined **for** t

define $\text{confl}' :: ('f, \text{nat} \times 's) \text{ term} \Rightarrow ('f, \text{nat} \times 's) \text{ term} \Rightarrow \text{nat} \times 's \Rightarrow \text{bool}$

where $\text{confl}' = (\lambda \text{ sp } tp \ y.$

$\text{sp} = \text{Var } y \wedge \text{inf-sort } (\text{snd } y) \wedge \text{sp} \neq \text{tp}$)

define $P1$ **where** $P1 = (\lambda \text{ mp } s \ t \ x \ y \ p. \text{mp} \in \text{pp} \longrightarrow (s, \text{Var } x) \in \text{mp} \wedge (t,$

$\text{Var } x) \in mp \wedge p \in \text{poss } s \wedge p \in \text{poss } t \wedge \text{confl}'(s \mid - p)(t \mid - p) y)$
 $\{$
 fix mp
 assume $mp \in pp$
 hence $\text{inf-var-conflict } mp$ **using** $*$ **by** auto
 from $\text{inf-var-conflictD}[OF \text{ this}]$
 have $\exists s t x y p. P1 mp s t x y p$ **unfolding** $P1\text{-def } \text{confl}'\text{-def}$ **by** force
 $\}$
hence $\forall mp. \exists s t x y p. P1 mp s t x y p$ **unfolding** $P1\text{-def}$ **by** blast
from $\text{choice}[OF \text{ this}]$ **obtain** s **where** $\forall mp. \exists t x y p. P1 mp (s mp) t x y$
 p **by** blast
 from $\text{choice}[OF \text{ this}]$ **obtain** t **where** $\forall mp. \exists x y p. P1 mp (s mp) (t mp)$
 $x y p$ **by** blast
 from $\text{choice}[OF \text{ this}]$ **obtain** x **where** $\forall mp. \exists y p. P1 mp (s mp) (t mp) (x$
 $mp) y p$ **by** blast
 from $\text{choice}[OF \text{ this}]$ **obtain** y **where** $\forall mp. \exists p. P1 mp (s mp) (t mp) (x$
 $mp) (y mp) p$ **by** blast
 from $\text{choice}[OF \text{ this}]$ **obtain** p **where** $\forall mp. P1 mp (s mp) (t mp) (x mp)$
 $(y mp) (p mp)$ **by** blast
 note $P1 = \text{this}[\text{unfolded } P1\text{-def, rule-format}]$
 from $*(2)$ **have** $\text{finite } (y \text{ ' } pp)$ **by** blast
 from $\text{ex-bij-betw-finite-nat}[OF \text{ this}]$ **obtain** index **and** $n :: \text{nat}$ **where**
 $\text{bij: bij-betw index } (y \text{ ' } pp) \{..<n\}$
 by $(\text{auto simp add: atLeast0LessThan})$
 define $\text{var-ind} :: \text{nat} \Rightarrow \text{nat} \times 's \Rightarrow \text{bool}$ **where**
 $\text{var-ind } i x = (x \in y \text{ ' } pp \wedge \text{index } x \in \{..<n\} - \{..<i\})$ **for** $i x$
 have $[\text{simp}]: \text{var-ind } n x = \text{False}$ **for** x
 unfolding var-ind-def **by** auto
 define $\text{cg-subst-ind} :: \text{nat} \Rightarrow ('f, \text{nat} \times 's)\text{subst} \Rightarrow \text{bool}$ **where**
 $\text{cg-subst-ind } i \sigma = (\forall x. (\text{var-ind } i x \longrightarrow \sigma x = \text{Var } x)$
 $\wedge (\neg \text{var-ind } i x \longrightarrow (\text{vars-term } (\sigma x) = \{\}) \wedge (\text{snd } x \in S \longrightarrow \sigma x : \text{snd}$
 $x \text{ in } \mathcal{T}(C, \emptyset)))$
 $\wedge (\text{snd } x \in S \longrightarrow \neg \text{inf-sort } (\text{snd } x) \longrightarrow \sigma x = \text{conv } (\delta x))$ **for** $i \sigma$
 define $\text{confl} :: \text{nat} \Rightarrow ('f, \text{nat} \times 's)\text{term} \Rightarrow ('f, \text{nat} \times 's)\text{term} \Rightarrow \text{bool}$ **where**
 $\text{confl} = (\lambda i sp tp.$
 $(\text{case } (sp, tp) \text{ of } (\text{Var } x, \text{Var } y) \Rightarrow x \neq y \wedge \text{var-ind } i x \wedge \text{var-ind } i y$
 $\mid (\text{Var } x, \text{Fun } -) \Rightarrow \text{var-ind } i x$
 $\mid (\text{Fun } -, \text{Var } x) \Rightarrow \text{var-ind } i x$
 $\mid (\text{Fun } f ss, \text{Fun } g ts) \Rightarrow (f, \text{length } ss) \neq (g, \text{length } ts))$
 have $\text{confl-n: confl } n s t \Longrightarrow \exists f g ss ts. s = \text{Fun } f ss \wedge t = \text{Fun } g ts \wedge$
 $(f, \text{length } ss) \neq (g, \text{length } ts)$ **for** $s t$
 by $(\text{cases } s; \text{cases } t; \text{auto simp: confl-def})$
 $\{$
 fix $i x$
 assume $\text{var-ind } i x$
 from $\text{this}[\text{unfolded } \text{var-ind-def}]$ **obtain** i
 where $z: x \in y \text{ ' } pp$ $\text{index } x = i$ **by** blast
 from z **obtain** mp **where** $mp \in pp$ **and** $\text{index } (y mp) = i$ **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 Term-empty-vars)
      show ¬ var-ind 0 x ⇒ snd x ∈ S ⇒ σ x : snd x in T(C,∅)
        using δ[THEN sorted-mapD, of x]
    unfolding σ-def conv-def by (auto simp: σ-def intro: subst-Term-empty-hastype)
    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))
          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  $\mathcal{T}(C, \emptyset :: \text{nat} \times 's \rightarrow)$  ∧ 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  $z\text{-inf}(1)$  [of  $d$ ]
obtain  $u :: (f, \text{nat} \times 's)$  term
  where  $u: u : \text{snd } z \text{ in } \mathcal{T}(C, \emptyset)$  and  $du: d \leq \text{size } u$  by auto
have  $\text{vars-}u: \text{vars } u = \{\}$  by (rule Term-empty-vars [OF  $u$ ])
define  $\sigma'$  where  $\sigma' x = (\text{if } x = z \text{ then } u \text{ else } \sigma x)$  for  $x$ 
have  $\sigma'\text{-def}' : \sigma' x = (\text{if } x \in y \text{ ' } pp \wedge \text{index } x = i \text{ then } u \text{ else } \sigma x)$  for  $x$ 
  unfolding  $\sigma'\text{-def}$  by (rule if-cong, insert bij z, auto simp: bij-betw-def inj-on-def)
have  $\text{var-ind-conv}: \text{var-ind } i x = (x = z \vee \text{var-ind } (\text{Suc } i) x)$  for  $x$ 
proof
  assume  $x = z \vee \text{var-ind } (\text{Suc } i) x$ 
  thus  $\text{var-ind } i x$  using  $z i$  unfolding var-ind-def by auto
next
  assume  $\text{var-ind } i x$ 
  hence  $x: x \in y \text{ ' } pp \text{ index } x \in \{..<n\} - \{..<i\}$  unfolding var-ind-def
by auto
  with  $i$  have  $\text{index } x = i \vee \text{index } x \in \{..<n\} - \{..<\text{Suc } i\}$  by auto
  thus  $x = z \vee \text{var-ind } (\text{Suc } i) x$ 
  proof
    assume  $\text{index } x = i$ 
    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]:  $\text{var-ind } i z$  unfolding var-ind-conv by auto
have [simp]:  $\text{var-ind } (\text{Suc } i) z = \text{False}$  unfolding var-ind-def using  $z$  by
auto
have  $\sigma z$  [simp]:  $\sigma z = \text{Var } z$  using  $\sigma$  [unfolded cg-subst-ind-def, rule-format, of z] by auto
have  $\sigma'\text{-upd}: \sigma' = \sigma(z := u)$  unfolding  $\sigma'\text{-def}$  by (intro ext, auto)
have  $\sigma'\text{-comp}: \sigma' = \sigma \circ_s \text{Var}(z := u)$  unfolding subst-compose-def  $\sigma'\text{-upd}$ 
proof (intro ext)
  fix  $x$ 
  show  $(\sigma(z := u)) x = \sigma x \cdot \text{Var}(z := u)$ 
  proof (cases x = z)
    case False
    hence  $\sigma x \cdot (\text{Var}(z := u)) = \sigma x \cdot \text{Var}$ 
    proof (intro term-subst-eq)
      fix  $y$ 
      assume  $y: y \in \text{vars } (\sigma x)$ 
      show  $(\text{Var}(z := u)) y = \text{Var } y$ 
      proof (cases var-ind i x)
        case True
        with  $\sigma$  [unfolded cg-subst-ind-def, rule-format, of x]
        have  $\sigma x = \text{Var } x$  by auto
        with False y show ?thesis by auto
      next
      case False
      with  $\sigma$  [unfolded cg-subst-ind-def, rule-format, of x]

```

```

      have vars ( $\sigma x$ ) = {} by auto
      with y show ?thesis by auto
    qed
  qed
  thus ?thesis by auto
  qed simp
  qed
  have  $\sigma'$ : cg-subst-ind (Suc i)  $\sigma'$  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  $x \neq i$  unfolding var-ind-def by auto
    hence  $\sigma x = \text{Var } x$  using  $\sigma$ [unfolded cg-subst-ind-def] by blast
    thus  $\sigma' x = \text{Var } x$  unfolding  $\sigma'$ -def' using diff by auto
  next
    fix x
    assume  $\neg$  var-ind (Suc i) x and snd  $x \in S$ 
    thus  $\sigma' x : \text{snd } x \text{ in } \mathcal{T}(C, \emptyset)$ 
      using  $\sigma$ [unfolded cg-subst-ind-def, rule-format, of x] u
      unfolding  $\sigma'$ -def var-ind-conv by auto
  next
    fix x
    assume  $\neg$  var-ind (Suc i) x
    hence  $x = z \vee \neg$  var-ind i x unfolding var-ind-conv by auto
    thus vars ( $\sigma' x$ ) = {} unfolding  $\sigma'$ -upd using  $\sigma$ [unfolded cg-subst-ind-def,
rule-format, of x] vars-u by auto
  next
    fix x :: nat  $\times$  's
    assume *: snd  $x \in S \neg$  inf-sort (snd x)
    with z-inf(2) have  $x \neq z$  by auto
    hence  $\sigma' x = \sigma x$  unfolding  $\sigma'$ -def by auto
    thus  $\sigma' x = \text{conv } (\delta x)$  using  $\sigma$ [unfolded cg-subst-ind-def, rule-format,
of x] * by auto
  qed
  show ?case
  proof (intro exI[of -  $\sigma'$ ] conjI  $\sigma'$  ballI)
    fix mp
    assume mp: mp  $\in$  pp
    define s' where s' = s mp  $\cdot$   $\sigma$ 
    define t' where t' = t mp  $\cdot$   $\sigma$ 
    from confl[rule-format, OF mp]
    obtain p where p: p  $\in$  poss s' p  $\in$  poss t' and confl: confl i (s' |- p) (t'
|- p) by (auto simp: s'-def t'-def)
    {
      fix s' t' :: ('f, nat  $\times$  's) term and p f ss x
      assume *: (s' |- p, t' |- p) = (Fun f ss, Var x) var-ind i x and p: p  $\in$ 
poss s' p  $\in$  poss t'
      and range-all-st: s'  $\in$  all-st
      hence s': s'  $\cdot$  Var(z := u) |- p = Fun f ss  $\cdot$  Var(z := u) (is - = ?s)

```

and $t': t' \cdot \text{Var}(z := u) \mid - p = (\text{if } x = z \text{ then } u \text{ else } \text{Var } x)$ **using** p
by *auto*
from *range-all-st[unfolded all-st-def]*
have $\text{range}\sigma: \exists S. s' = S \cdot \sigma$ **by** *auto*
define s **where** $s = ?s$
have $\exists p. p \in \text{poss } (s' \cdot \text{Var}(z := u)) \wedge p \in \text{poss } (t' \cdot \text{Var}(z := u)) \wedge$
confl (Suc i) (s' \cdot \text{Var}(z := u) \mid - p) (t' \cdot \text{Var}(z := u) \mid - p)
proof (*cases* $x = z$)
case *False*
thus *?thesis* **using** $* p$ **unfolding** $s' t'$ **by** (*intro exI[of - p]*, *auto simp: confl-def var-ind-conv*)
next
case *True*
hence $t': t' \cdot \text{Var}(z := u) \mid - p = u$ **unfolding** t' **by** *auto*
have $\exists p'. p' \in \text{poss } u \wedge p' \in \text{poss } s \wedge \text{confl } (Suc i) (s \mid - p') (u \mid - p')$
proof (*cases* $\exists x. x \in \text{vars } s \wedge \text{var-ind } (Suc i) x$)
case *True*
then obtain x **where** $xs: x \in \text{vars } s$ **and** $x: \text{var-ind } (Suc i) x$ **by**
auto
from xs **obtain** p' **where** $p': p' \in \text{poss } s$ **and** $sp: s \mid - p' = \text{Var } x$
by (*metis vars-term-poss-subt-at*)
from $p' sp \text{ vars-u}$ **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** $s: s = \text{Var } x$ **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** $p: p' = j \# p j < \text{length } ss p \in \text{poss}$
 $(ss ! j) (ss ! j) \mid - p = \text{Var } x$ **by** *auto*
show *?thesis*
proof (*cases* $(f, \text{length } us) = (g, \text{length } ss)$)
case *False*
thus *?thesis* **by** (*intro exI[of - Nil]*, *auto simp: s confl-def*)
next
case *True*
with p **have** $j: j < \text{length } us$ **by** *auto*
hence $usj: us ! j \in \text{set } us$ **by** *auto*
with *Fun* **have** $\text{vars } (us ! j) = \{\}$ **by** *auto*
from *Fun(1)[OF usj p(3,4) this]* **obtain** p' **where**
 $p' \in \text{poss } (us ! j) \wedge p' \in \text{poss } (ss ! j) \wedge \text{confl } (Suc i) (ss ! j \mid -$
 $p') (us ! j \mid - p')$ **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-subst-at)
also have ... =  $(\bigcup_{y \in \text{vars } S} \text{vars } (\sigma y))$  unfolding sS by (simp
add: vars-term-subst)
also have ...  $\subseteq (\bigcup_{y \in \text{vars } S} \text{Collect } (\text{var-ind } i))$ 
proof -
{
  fix x y
  assume x ∈ vars (σ y)
  hence var-ind i x
  using  $\sigma[\text{unfolded } \text{cg-subst-ind-def}, \text{rule-format}, \text{of } y]$  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 ... =  $\bigcup (\text{vars } \langle \text{Var}(z := u) \rangle \langle \text{vars } (s' |- p) \rangle)$  by (simp
add: vars-term-subst)
also have ...  $\subseteq \bigcup (\text{vars } \langle \text{Var}(z := u) \rangle \langle \text{Collect } (\text{var-ind } i) \rangle)$  using
sub by auto
also have ...  $\subseteq \text{Collect } (\text{var-ind } (\text{Suc } i))$ 
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-subst-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 $p \in \text{poss } (t' \cdot \text{Var}(z := u))$ **using** p **by** *auto*
moreover have $p' \in \text{poss } ((t' \cdot \text{Var}(z := u)) \mid - p)$ **using** $t' p' p$ **by**
auto
ultimately show $p @ p' \in \text{poss } (t' \cdot \text{Var}(z := u))$ **by** *simp*
qed
qed
} **note** $\text{main} = \text{this}$
consider $(FF) f g ss ts$ **where** $(s' \mid - p, t' \mid - p) = (\text{Fun } f ss, \text{Fun } g ts)$
 $(f, \text{length } ss) \neq (g, \text{length } ts)$
 $\mid (FV) f ss x$ **where** $(s' \mid - p, t' \mid - p) = (\text{Fun } f ss, \text{Var } x)$ *var-ind i x*
 $\mid (VF) f ss x$ **where** $(s' \mid - p, t' \mid - p) = (\text{Var } x, \text{Fun } f ss)$ *var-ind i x*
 $\mid (VV) x x'$ **where** $(s' \mid - p, t' \mid - p) = (\text{Var } x, \text{Var } x')$ $x \neq x'$ *var-ind i x'*
using *confl* **by** $(\text{auto } \text{simp}: \text{confl-def split}: \text{term.splits})$
hence $\exists p. p \in \text{poss } (s' \cdot \text{Var}(z := u)) \wedge p \in \text{poss } (t' \cdot \text{Var}(z := u)) \wedge$
 $\text{confl } (Suc i) (s' \cdot \text{Var}(z := u) \mid - p) (t' \cdot \text{Var}(z := u) \mid - p)$
proof *cases*
case $(FF) f g ss ts$
thus *?thesis* **using** p **by** $(\text{intro } \text{exI}[\text{of } - p], \text{auto } \text{simp}: \text{confl-def})$
next
case $(FV) f ss x$
have $s' \in \text{all-st}$ **unfolding** s' -def **using** mp all-st-def **by** *auto*
from $\text{main}[\text{OF } FV p \text{ this}]$ **show** *?thesis* **by** *auto*
next
case $(VF) f ss x$
have $t': t' \in \text{all-st}$ **unfolding** t' -def **using** mp all-st-def **by** *auto*
from VF **have** $(t' \mid - p, s' \mid - p) = (\text{Fun } f ss, \text{Var } x)$ *var-ind i x* **by** *auto*
from $\text{main}[\text{OF } \text{this } p(2,1) t']$
obtain p **where** $p \in \text{poss } (t' \cdot \text{Var}(z := u)) \wedge p \in \text{poss } (s' \cdot \text{Var}(z := u))$
 $\text{confl } (Suc i) (t' \cdot \text{Var}(z := u) \mid - p) (s' \cdot \text{Var}(z := u) \mid - p)$
by *auto*
thus *?thesis* **by** $(\text{intro } \text{exI}[\text{of } - p], \text{auto } \text{simp}: \text{confl-def split}: \text{term.splits})$
next
case $(VV) x x'$
thus *?thesis* **using** p *vars-u* **by** $(\text{intro } \text{exI}[\text{of } - p], \text{cases } u, \text{auto } \text{simp}: \text{confl-def var-ind-conv})$
qed
thus $\exists p. p \in \text{poss } (s \text{ mp} \cdot \sigma') \wedge p \in \text{poss } (t \text{ mp} \cdot \sigma') \wedge \text{confl } (Suc i) (s$
 $\text{mp} \cdot \sigma' \mid - p) (t \text{ mp} \cdot \sigma' \mid - p)$
unfolding σ' -comp *subst-subst-compose* s' -def t' -def **by** *auto*
qed
qed
}
from $\text{this}[\text{of } n]$
obtain σ **where** $\sigma: \text{cg-subst-ind } n \sigma$ **and** *confl*: $\bigwedge \text{mp}. \text{mp} \in \text{pp} \implies \exists p. p \in$
 $\text{poss } (s \text{ mp} \cdot \sigma) \wedge p \in \text{poss } (t \text{ mp} \cdot \sigma) \wedge \text{confl } n (s \text{ mp} \cdot \sigma \mid - p) (t \text{ mp} \cdot \sigma \mid - p)$
by *blast*
define $\sigma' :: (f, \text{nat} \times 's, \text{unit}) \text{gsubst}$ **where** $\sigma' x = \text{conv}' (\text{Var } x)$ **for** x
let $?\sigma = \sigma \circ_s \sigma'$

```

{
  fix x :: nat × 's
  assume *: snd x ∈ S ¬ inf-sort (snd x)
  from δ[THEN sorted-mapD, of x] * have δ x : snd x in T(C,∅) by auto
  hence vars: vars (δ x) = {} by (simp add: Term-empty-vars)
  from * σ[unfolded cg-subst-ind-def] have σ x = conv (δ x) by blast
  hence ?σ x = δ x · (undefined ∘s σ') by (simp add: subst-compose-def
conv-def subst-subst)
  also have ... = δ x by (rule ground-term-subst[OF vars])
  finally have ?σ x = δ x .
} note σδ = this
have ?σ :s {x : ι in V. ι ∈ S} → 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 : ι in V and ι ∈ S
  then have snd x = ι ι ∈ S by auto
  with σ[unfolded cg-subst-ind-def, rule-format, of x]
  have σ x : ι in T(C,∅) by auto
  thus σ x · σ' : ι in T(C,∅) by (rule subst-Term-empty-hastype)
qed
from pp[unfolded wf-pat-complete-iff[OF wf] match-complete-wrt-def, rule-format,
OF this]
  obtain mp μ where mp: mp ∈ pp ∪ pp' and match: ∧ ti li. (ti, li) ∈ mp ⇒
ti · ?σ = li · μ 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
next
case (pp-success pp)
then show ?case by (auto simp: pat-complete-def match-complete-wrt-def)
next
case *: (pp-instantiate n pp x)
have wfpp: wf-pat pp by fact

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
  assume complete: Ball P' (pat-complete C)
  show pat-complete C pp unfolding def[OF wfpp]

```

```

proof (intro allI impI)
  fix  $\sigma$ 
  assume  $cg: \sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$ 
  from sorted-mapD[OF this]  $x$ 
  have  $\sigma x : \text{snd } x \text{ in } \mathcal{T}(C)$  by auto
  then obtain  $f \text{ ts } \sigma s$  where  $f: f : \sigma s \rightarrow \text{snd } x \text{ in } C$ 
    and  $args: ts :_\iota \sigma s \text{ in } \mathcal{T}(C)$ 
    and  $\sigma x: \sigma x = \text{Fun } f \text{ ts}$ 
    by (induct, auto)
  from  $f$  have  $f: f : \sigma s \rightarrow \text{snd } x \text{ in } C$ 
    by (meson fun-hastype-def)
  let  $?l = \text{length } ts$ 
  from  $args$  have  $len: \text{length } \sigma s = ?l$  by (simp add: list-all2-lengthD)
  have  $l: ?l \leq m$  using  $m[OF f] \text{ len}$  by auto
  have  $\sigma s S: \forall \iota \in \text{set } \sigma s. \iota \in S$  using  $C\text{-sub-}S f$  by auto
  define  $\sigma'$  where  $\sigma' = (\lambda \text{ ys}. \text{let } y = \text{fst } \text{ys} \text{ in if } n \leq y \wedge y < n + ?l \wedge \sigma s ! (y - n) = \text{snd } \text{ys} \text{ then } ts ! (y - n) \text{ else } \sigma \text{ ys})$ 
  have  $cg: \sigma' :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$ 
  proof (intro sorted-mapI, unfold hastype-in-restrict-sset conj-imp-eq-imp-imp)
    fix  $\text{ys} :: \text{nat} \times 's$  and  $\iota$ 
    assume  $\text{ys} : \iota \text{ in } \mathcal{V}$  and  $\iota \in S$ 
    then have  $[simp]: \iota = \text{snd } \text{ys}$  and  $\text{ys} S: \text{snd } \text{ys} \in S$  by auto
    show  $\sigma' \text{ ys} : \iota \text{ in } \mathcal{T}(C)$ 
    proof (cases  $\sigma' \text{ ys} = \sigma \text{ ys}$ )
      case True
        thus  $?thesis$  using  $cg \text{ ys} S$  by (auto simp: sorted-mapD)
      next
        case False
          obtain  $y \text{ s}$  where  $\text{ys}: \text{ys} = (y, s)$  by force
          with False have  $y: y - n < ?l \wedge n \leq y \wedge y < n + ?l$  and  $\text{arg}: \sigma s ! (y - n)$ 
            and  $\sigma': \sigma' \text{ ys} = ts ! (y - n)$ 
            unfolding  $\sigma'\text{-def } \text{Let-def}$  by (auto split: if-splits)
          show  $?thesis$ 
            using  $\sigma' \text{ len } \text{list-all2-nthD}[OF \text{ args } y(1)]$ 
            by (auto simp:  $\text{ys arg[symmetric]}$ )
    qed
  qed
  define  $\tau$  where  $\tau = \text{subst } x (\text{Fun } f (\text{map } \text{Var} (\text{indexed-from } n \sigma s)))$ 
  have  $\tau :_s \mathcal{V} \mid^t \text{tvars-pat } pp \rightarrow \mathcal{T}(C, \{x : \iota \text{ in } \mathcal{V}. \iota \in S\})$ 
    using  $\text{Fun-hastypeI}[OF f, of \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \text{ map } \text{Var} (\text{indexed-from } n \sigma s)] \sigma s S \text{ wfpp}$ 
    by (auto intro!: sorted-mapI
      simp:  $\tau\text{-def } \text{subst-def } \text{list-all2-conv-all-nth } \text{hastype-restrict } \text{wf-pat-iff}$ 
       $\text{nth-indexed-from-eq } \text{simp flip: len}$ )
  from  $\text{wf-pat-subst}[OF \text{ this}]$ 
  have  $\text{wf2}: \text{wf-pat} (\text{subst-pat-problem-set } \tau \text{ pp})$ .
  from  $f$  have  $\tau \in \tau s \text{ n } x$  unfolding  $\tau s\text{-def } \tau\text{-def } \tau c\text{-def}$  using  $\text{len[symmetric]}$ 
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 ∈ subst-pat-problem-set τ pp
    and match:  $\bigwedge ti li. (ti, li) \in tl \implies ti \cdot \sigma' = li \cdot \mu$  by force
  from tl[unfolded subst-defs-set subst-left-def set-map]
  obtain tl' where tl': tl' ∈ pp and tl: tl =  $\{(t' \cdot \tau, l) \mid (t', l) \in tl'\}$  by auto
  show  $\exists tl \in pp. \exists \mu. \forall (ti, li) \in tl. ti \cdot \sigma = li \cdot \mu$ 
  proof (intro bexI[OF - tl'] exI[of - μ], clarify)
    fix ti li
    assume tli:  $(ti, li) \in tl'$ 
    hence tlit:  $(ti \cdot \tau, li) \in tl$  unfolding tl by force
    from match[OF this] have match:  $ti \cdot \tau \cdot \sigma' = li \cdot \mu$  by auto
    from *(1)[unfolded tvars-disj-pp-def, rule-format, OF tli]
    have vti:  $fst \text{ ` vars-term } ti \cap \{n..<n + m\} = \{\}$  by auto
    have  $ti \cdot \sigma = ti \cdot (\tau \circ_s \sigma')$ 
    proof (rule term-subst-eq, unfold subst-compose-def)
      fix y
      assume y ∈ vars-term ti
      with vti have y:  $fst y \notin \{n..<n + m\}$  by auto
      show  $\sigma y = \tau y \cdot \sigma'$ 
      proof (cases y = x)
        case False
          hence  $\tau y \cdot \sigma' = \sigma' y$  unfolding τ-def subst-def by auto
          also have  $\dots = \sigma y$ 
            unfolding σ'-def using y l by auto
          finally show ?thesis by simp
        case True
          have  $\langle ts = map \sigma' (indexed-from n \sigma s) \rangle$ 
            by (rule nth-equalityI) (simp-all add: len nth-indexed-from-eq σ'-def)
          with True σx show ?thesis by (simp add: τ-def o-def)
      qed
    qed
    also have  $\dots = li \cdot \mu$  using match by simp
    finally show  $ti \cdot \sigma = li \cdot \mu$  by blast
  qed
next
  assume complete: pat-complete C pp
  show  $\forall pp \in P'. pat-complete C pp$ 
  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:  $f : \iota s \rightarrow snd x$  in C and τ:  $\tau = subst x (Fun f (map$ 
Var (indexed-from n ιs)))
    by auto

```

```

let ?i = length  $\iota$ s
let ?xs = indexed-from n  $\iota$ s
have i: ?i ≤ m by (rule m[OF f])
have  $\forall \iota \in \text{set } \iota$ s.  $\iota \in S$  using C-sub-S f by blast
with Fun-hastypeI[OF f, of {x :  $\iota$  in  $\mathcal{V}$ .  $\iota \in S$ } map Var ?xs] wfpp
have  $\tau :_s \mathcal{V} \mid^{\cdot} \text{tvars-pat } pp \rightarrow \mathcal{T}(C, \{x : \iota \text{ in } \mathcal{V}. \iota \in S\})$ 
  by (auto intro!: sorted-mapI
      simp:  $\tau$  subst-def hastype-restrict list-all2-conv-all-nth wf-pat-iff nth-indexed-from-eq)
note def2 = def[OF wf-pat-subst[OF this]]
show pat-complete C (subst-pat-problem-set  $\tau$  pp) unfolding def2
proof (intro allI impI)
  fix  $\sigma$  assume cg:  $\sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$ 
  define  $\sigma'$  where  $\sigma' = \sigma(x := \text{Fun } f \text{ (map } \sigma \text{ ?xs)})$ 
  from C-sub-S[OF f] have sortsS: set  $\iota$ s ⊆ S by auto
  from f have f: f :  $\iota$ s → snd x in C by (simp add: fun-hastype-def)
  with sorted-mapD[OF cg] set-mp[OF sortsS]
  have Fun f (map  $\sigma$  ?xs) : snd x in  $\mathcal{T}(C)$ 
  by (auto intro!: Fun-hastypeI simp: list-all2-conv-all-nth nth-indexed-from-eq)
  with sorted-mapD[OF cg]
  have cg:  $\sigma' :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$  by (auto intro!: sorted-mapI
simp:  $\sigma'$ -def)
  from complete[unfolded def[OF wfpp], rule-format, OF this]
  obtain tl  $\mu$  where tl: tl ∈ pp and tli:  $\bigwedge ti \ li. (ti, li) \in tl \implies ti \cdot \sigma' = li \cdot \mu$  by force
  from tl have tlm:  $\{(t \cdot \tau, l) \mid (t, l) \in tl\} \in \text{subst-pat-problem-set } \tau \text{ pp}$ 
  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  $\iota$  vars-term ti ∩
{n.. $n + m$ } = {} by force
    from tli[OF mem] have li ·  $\mu = ti \cdot \sigma'$  by auto
    also have ... = ti · ( $\tau \circ_s \sigma$ )
    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  $\sigma' y = \tau y \cdot \sigma$ 
      proof (cases y = x)
        case False
          hence  $\tau y \cdot \sigma = \sigma y$  unfolding  $\tau$  subst-def by auto
          also have ... =  $\sigma' y$ 
          unfolding  $\sigma'$ -def using False by auto
        finally show ?thesis by simp
      case True
        show ?thesis unfolding True  $\tau$ 
          by (simp add: o-def  $\sigma'$ -def)
    qed
  }
qed

```

```

      qed
    finally have  $ti \cdot \tau \cdot \sigma = li \cdot \mu$  by auto
  }
  thus  $\exists tl \in \text{subst-pat-problem-set } \tau \text{ pp. } \exists \mu. \forall (ti, li) \in tl. ti \cdot \sigma = li \cdot \mu$ 
    by (intro bexF[OF - tlm], auto)
  qed
qed
qed
qed

```

theorem *P-step-set-pcorrect:*

$P \Rightarrow_s P' \implies \text{wf-pats } P \implies \text{pats-complete } C P \longleftrightarrow \text{pats-complete } C P'$

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* $pp P' P$)

with *pp-step-wf* have *wf-pat* pp *wf-pats* P *wf-pats* (*insert* $pp P$) *wf-pats* ($P' \cup P$)

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*)

qed

end

Represent a variable-form as a set of maps.

definition *match-of-var-form* $f = \{(Var\ y, Var\ x) \mid x\ y. y \in f\ x\}$

definition *pat-of-var-form* $ff = \text{match-of-var-form } 'ff$

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

definition *var-form-of-pat* $pp = \text{var-form-of-match } 'pp$

definition *tvars-var-form-pat* $ff = (\bigcup f \in ff. \bigcup (\text{range } f))$

definition *var-form-match* **where**

var-form-match $mp \longleftrightarrow mp \subseteq \text{range } (\text{map-prod } Var\ Var)$

definition *var-form-pat* $pp \equiv \forall mp \in pp. \text{var-form-match } mp$

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* $mp = \{v. \exists x. (Var\ v, Var\ x) \in mp\}$

using *assms* by (force simp: var-form-match-def tvar-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) = tvar-var-form-pat ff*

by (fastforce simp: tvar-var-form-pat-def tvar-pat-def tvar-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) = tvar-pat pp*

apply (subst (2) pat-of-var-form-pat[OF *assms*,symmetric])

by (simp add: tvar-pat-var-form)

lemma *pat-complete-var-form*:

pat-complete C (pat-of-var-form ff) \longleftrightarrow

($\forall \sigma :_s \mathcal{V} \mid \text{tvars-var-form-pat ff} \rightarrow \mathcal{T}(C). \exists f \in \text{ff}. \exists \mu. \forall x. \forall y \in f x. \sigma y = \mu x$)

proof –

define *V* where *V = $\mathcal{V} \mid \text{tvars-var-form-pat ff}$*

have *boo: $\mathcal{V} \mid \text{tvars-pat} \{ \{ (Var (a, b), Var xa) \mid xa a b. (a, b) \in x xa \} \mid x \in \text{ff} \}$*

= V

apply (unfold *V-def*)

apply (subst tvar-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) \longleftrightarrow

($\forall \sigma :_s \mathcal{V} \mid \text{tvars-var-form-pat ff} \rightarrow \mathcal{T}(C). \exists f \in \text{ff}. \exists \mu. \forall x. \sigma ' f x \subseteq \{ \mu x \}$)

by (auto simp: *pat-complete-var-form image-subset-iff*)

lemma *pat-complete-var-form-Uniq*:

pat-complete C (pat-of-var-form ff) \longleftrightarrow

($\forall \sigma :_s \mathcal{V} \mid \text{tvars-var-form-pat ff} \rightarrow \mathcal{T}(C). \exists f \in \text{ff}. \forall x. \text{UNIQ} (\sigma ' f x)$)

proof –

{ fix *σ f* assume *σ: σ :_s $\mathcal{V} \mid \text{tvars-var-form-pat ff} \rightarrow \mathcal{T}(C)$ and *f: f ∈ ff**

have *($\exists \mu. \forall x. \sigma ' f x \subseteq \{ \mu x \}$) \longleftrightarrow ($\forall x. \exists_{\leq 1} y. y \in \sigma ' f x$)*

```

proof (safe)
  fix  $\mu$   $x$ 
  assume  $\forall x. \sigma \text{ ' } f x \subseteq \{\mu x\}$ 
  from this[rule-format, of x]
  have  $y \in f x \implies \sigma y = \mu x$  for  $y$  by auto
  then show  $\exists_{\leq 1} y. y \in \sigma \text{ ' } f x$  by (auto intro!: Uniq-I)
next
  define  $\mu$  where  $\mu x = \text{the-elem } (\sigma \text{ ' } f x)$  for  $x$ 
  fix  $x$  assume  $\forall x. \exists_{\leq 1} y. y \in \sigma \text{ ' } f x$ 
  from Uniq-eq-the-elem[OF this[rule-format], folded  $\mu$ -def]
  show  $\exists \mu. \forall x. \sigma \text{ ' } f x \subseteq \{\mu x\}$  by auto
qed
}
then show ?thesis by (simp add: pat-complete-var-form-set)
qed

lemma ex-var-form-pat:  $(\exists f \in \text{var-form-of-pat } pp. P f) \longleftrightarrow (\exists mp \in pp. P (\text{var-form-of-match } mp))$ 
by (auto simp: var-form-of-pat-def)

lemma pat-complete-var-form-nat:
assumes fin:  $\forall (x,\iota) \in \text{tvars-var-form-pat } ff. \text{finite-sort } C \iota$ 
and uniq:  $\forall f \in ff. \forall x::'v. \text{UNIQ } (\text{snd } \text{' } f x)$ 
shows pat-complete  $C$  (pat-of-var-form  $ff$ )  $\longleftrightarrow$ 
 $(\forall \alpha. (\forall v \in \text{tvars-var-form-pat } ff. \alpha v < \text{card-of-sort } C (\text{snd } v)) \longrightarrow$ 
 $(\exists f \in ff. \forall x. \text{UNIQ } (\alpha \text{ ' } f x)))$ 
(is ?l  $\longleftrightarrow (\forall \alpha. ?s \alpha \longrightarrow ?r \alpha)$ 
proof safe
note fin = fin[unfolded Ball-Pair-conv, rule-format]
{ fix  $\alpha$ 
assume l: ?l and a: ?s  $\alpha$ 
define  $\sigma :: - \Rightarrow (-, \text{unit}) \text{ term where}$ 
 $\sigma \equiv \lambda(x,\iota). \text{term-of-index } C \iota (\alpha (x,\iota))$ 
have  $\sigma (x,\iota) : \iota \text{ in } \mathcal{T}(C)$  if  $x: (x,\iota) \in \text{tvars-var-form-pat } ff$  for  $x \iota$ 
using term-of-index-bij[OF fin, OF x]
 $a$ [unfolded Ball-Pair-conv, rule-format, OF x]
by (auto simp: bij-betw-def  $\sigma$ -def)
then have  $\sigma :_s \mathcal{V} \mid \text{tvars-var-form-pat } ff \rightarrow \mathcal{T}(C)$ 
by (auto intro!: sorted-mapI simp: hastype-restrict)
from l[unfolded pat-complete-var-form-Uniq, rule-format, OF this]
obtain  $f$  where  $f: f \in ff$  and  $u: \bigwedge x. \text{UNIQ } (\sigma \text{ ' } f x)$  by auto
have id:  $y \in f x \implies \text{index-of-term } C (\sigma y) = \alpha y$  for  $y x$ 
using assms a f
by (force simp:  $\sigma$ -def index-of-term-of-index tvars-var-form-pat-def Ball-def
split: prod.splits)
then have  $\alpha \text{ ' } f x = \text{index-of-term } C \text{ ' } \sigma \text{ ' } f x$  for  $x$ 
by (auto simp: image-def)
then have UNIQ  $(\alpha \text{ ' } f x)$  for  $x$  by (simp add: image-Uniq[OF u])
with  $f$  show ?r  $\alpha$  by auto

```

```

next
  assume r:  $\forall \alpha. ?s \alpha \longrightarrow ?r \alpha$ 
  show ?l
    unfolding pat-complete-var-form-Uniq
  proof safe
    fix  $\sigma$ 
    assume  $\sigma: \sigma :_s \mathcal{V} \mid ' tvars\text{-var}\text{-form}\text{-pat} \text{ ff} \rightarrow \mathcal{T}(C)$ 
    from sorted-mapD[OF this]
    have ty:  $(x, \iota) \in tvars\text{-var}\text{-form}\text{-pat} \text{ ff} \implies \sigma (x, \iota) : \iota \text{ in } \mathcal{T}(C)$ 
      for  $x \iota$  by (auto simp: hastype-restrict)
    define  $\alpha$  where  $\alpha \equiv \text{index-of-term } C \circ \sigma$ 
    have  $\alpha (x, \iota) < \text{card-of-sort } C \iota$  if  $x: (x, \iota) \in tvars\text{-var}\text{-form}\text{-pat} \text{ ff}$ 
      for  $x \iota$  using index-of-term-bij[OF fin[OF x]] ty[OF x]
      by (auto simp:  $\alpha$ -def bij-betw-def)
    then have  $\exists f \in \text{ff}. \forall x. \text{UNIQ } (\alpha ' f x)$  by (auto intro!: r[rule-format])
    then obtain f where  $f: f \in \text{ff}$  and  $u: \bigwedge x. \text{UNIQ } (\alpha ' f x)$  by auto
    have UNIQ ( $\sigma ' f x$ ) for x
  proof-
    from uniq[rule-format, OF f]
    have ex:  $\exists \iota. \text{snd } ' f x \subseteq \{\iota\}$ 
      by (auto simp: subset-singleton-iff-Uniq)
    then obtain  $\iota$  where  $\text{sub}: \text{snd } ' f x \subseteq \{\iota\}$  by auto
    { fix y  $\kappa$  assume  $y\kappa: (y, \kappa) \in f x$ 
      with sub have [simp]:  $\kappa = \iota$  by auto
      from  $y\kappa f$  have  $y: (y, \iota) \in tvars\text{-var}\text{-form}\text{-pat} \text{ ff}$ 
        by (auto simp: tvars-var-form-pat-def)
      from y fin[OF y]
      have term-of-index  $C \iota (\alpha (y, \kappa)) = \sigma (y, \kappa)$ 
        by (auto simp:  $\alpha$ -def hastype-restrict
          intro!: term-of-index-of-term sorted-mapD[OF  $\sigma$ ])
    }
    then have  $y \in f x \implies \text{term-of-index } C \iota (\alpha y) = \sigma y$  for y
      by (cases y, auto)
    then have  $\sigma ' f x = \text{term-of-index } C \iota ' \alpha ' f x$ 
      by (auto simp: image-def)
    then show UNIQ ( $\sigma ' f x$ ) by (simp add: image-Uniq[OF u])
  qed
  with f show  $\exists f \in \text{ff}. \forall x. \text{UNIQ } (\sigma ' f x)$  by auto
qed
}
qed

end
theory FCF-Problem
  imports Pattern-Completeness-Set
begin

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

```

definition *UNIQ-subst* **where** $UNIQ\text{-subst } \sigma A = UNIQ (A \cdot_{set} \sigma)$

lemma *UNIQ-subst-pairI*: **assumes** $\bigwedge s t. s \in A \implies t \in A \implies s \cdot \sigma = t \cdot \sigma$
shows $UNIQ\text{-subst } \sigma A$ **unfolding** *UNIQ-subst-def* *Uniq-def* **using** *assms* **by** *blast*

lemma *UNIQ-subst-trivial[simp]*: $UNIQ\text{-subst } \sigma \{t\} UNIQ\text{-subst } \sigma \{\}$
by (*auto simp: UNIQ-subst-def Uniq-def*)

lemma *UNIQ-subst-pairD*: **assumes** $UNIQ\text{-subst } \sigma A$
shows $s \in A \implies t \in A \implies s \cdot \sigma = t \cdot \sigma$
using *assms* **unfolding** *UNIQ-subst-def* *Uniq-def* **by** *blast*

lemma *UNIQ-mono*: **assumes** $A \subseteq B$
shows $UNIQ B \implies UNIQ A$ **using** *assms*
by (*simp add: Uniq-def subset-iff*)

lemma *UNIQ-subst-mono*: **assumes** $A \subseteq B$
shows $UNIQ\text{-subst } \sigma B \implies UNIQ\text{-subst } \sigma A$
unfolding *UNIQ-subst-def* **using** *assms*
by (*metis UNIQ-mono image-mono*)

lemma *UNIQ-subst-alt-def*: $UNIQ\text{-subst } \sigma A = (\forall s t. s \in A \longrightarrow t \in A \longrightarrow s \cdot \sigma = t \cdot \sigma)$
unfolding *UNIQ-subst-def* *Uniq-def* **by** *auto*

definition *simple-match-complete-wrt* :: $(f, nat \times 's, 'w)gsubst \Rightarrow (f, 's)simple\text{-match}\text{-problem} \Rightarrow bool$ **where**
 $simple\text{-match}\text{-complete}\text{-wrt } \sigma mp = (\forall eqc \in mp. UNIQ\text{-subst } \sigma eqc)$

type-synonym $(f, 's)simple\text{-pat}\text{-problem} = (f, 's)simple\text{-match}\text{-problem set}$

abbreviation *tvars-spat* :: $(f, 's)simple\text{-pat}\text{-problem} \Rightarrow (nat \times 's) set$ **where**
 $tvars\text{-spat } spp \equiv \bigcup (\bigcup (\bigcup (image (image vars) ' spp)))$

abbreviation *tvars-smp* :: $(f, 's)simple\text{-match}\text{-problem} \Rightarrow (nat \times 's) set$ **where**
 $tvars\text{-smp } smp \equiv \bigcup (\bigcup (image vars ' smp))$

definition *simple-pat-complete* :: $(f, 's) sig \Rightarrow (nat \times 's) set \Rightarrow (f, 's)simple\text{-pat}\text{-problem} \Rightarrow bool$ **where**
 $simple\text{-pat}\text{-complete } C S pp \iff (\forall \sigma :_s \mathcal{V} \mid ' S \rightarrow \mathcal{T}(C). \exists mp \in pp. simple\text{-match}\text{-complete}\text{-wrt } \sigma mp)$

lemma *tvars-spat-cong*: **assumes** $\bigwedge x. x \in tvars\text{-spat } spp \implies \sigma x = \delta x$
and $mp \in spp$

shows $simple\text{-match}\text{-complete}\text{-wrt } \sigma mp = simple\text{-match}\text{-complete}\text{-wrt } \delta mp$
unfolding *simple-match-complete-wrt-def* *UNIQ-subst-alt-def*
apply (*intro ball-cong refl all-cong1 imp-cong*)
apply (*subst (1 2) term-subst-eq[of - $\sigma \delta$]*)

using *assms* **by** *force+*

abbreviation *set2* :: 'a list list \Rightarrow 'a set set **where** *set2* \equiv *image set o set*

abbreviation *set3* :: 'a list list list \Rightarrow 'a set set set **where** *set3* \equiv *image set2 o set*

context *pattern-completeness-context*
begin

definition *finite-constructor-form-mp* :: ('f,'s)*simple-match-problem* \Rightarrow bool **where**
finite-constructor-form-mp mp = (\forall *eqc* \in *mp*. *eqc* \neq {} \wedge (\exists ι . *finite-sort C* ι
 \wedge (\forall $t \in$ *eqc*. $t : \iota$ in $\mathcal{T}(C, \mathcal{V} \mid 'SS)$)))

definition *finite-constructor-form-pat* $p =$ *Ball p finite-constructor-form-mp*

lemmas *finite-constructor-form-defs* = *finite-constructor-form-pat-def finite-constructor-form-mp-def*

definition *fcf-solver* **where**
fcf-solver k solver = (\forall *fcf n*.
finite-constructor-form-pat (set3 fcf) \longrightarrow
tvars-spat (set3 fcf) \subseteq $\{..<n\} \times UNIV$ \longrightarrow
length fcf $<$ k \longrightarrow
solver n fcf = *simple-pat-complete C SS (set3 fcf)*)

end

end

6 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
FCF-Problem
begin

6.1 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 $(f, 'v, 's)match\text{-}problem\text{-}mset = ((f, nat \times 's)term \times (f, 'v)term)$
multiset

type-synonym $(f, 'v, 's)pat\text{-}problem\text{-}mset = (f, 'v, 's)match\text{-}problem\text{-}mset$ *multiset*

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

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

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

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

where $pat\text{-}mset \equiv image\ mp\text{-}mset\ o\ set\text{-}mset$

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

where $pats\text{-}mset \equiv image\ pat\text{-}mset\ o\ set\text{-}mset$

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

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

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

definition $\tau s\text{-}list :: nat \Rightarrow nat \times 's \Rightarrow (f, nat \times 's)subst\ list$ **where**
 $\tau s\text{-}list\ n\ x = map\ (\tau c\ n\ x)\ (Cl\ (snd\ x))$

inductive $mp\text{-}step\text{-}mset :: (f, 'v, 's)match\text{-}problem\text{-}mset \Rightarrow (f, 'v, 's)match\text{-}problem\text{-}mset$
 $\Rightarrow bool$ (**infix** $\langle \rightarrow_m \rangle$ 50) **where**

$match\text{-}decompose: (f, length\ ts) = (g, length\ ls)$
 $\implies add\text{-}mset\ (Fun\ f\ ts,\ Fun\ g\ ls)\ mp \rightarrow_m mp + mset\ (zip\ ts\ ls)$

| $match\text{-}match: x \notin \bigcup (vars\ 'snd\ 'set\text{-}mset\ mp)$
 $\implies add\text{-}mset\ (t,\ Var\ x)\ mp \rightarrow_m mp$

| $match\text{-}duplicate: add\text{-}mset\ pair\ (add\text{-}mset\ pair\ mp) \rightarrow_m add\text{-}mset\ pair\ mp$

| $match\text{-}decompose': mp + mp' \rightarrow_m (\sum (t, l) \in \# mp. mset\ (zip\ (args\ t)\ (map\ Var\ ys))) + mp'$

if $\bigwedge t\ l. (t, l) \in \# mp \implies l = Var\ y \wedge root\ t = Some\ (f, n)$
 $\bigwedge t\ l. (t, l) \in \# mp' \implies y \notin vars\ l$
 $lvars\text{-}disj\text{-}mp\ ys\ (mp\text{-}mset\ (mp + mp'))\ length\ ys = n$
 $size\ mp \geq 2$

improved

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

inductive *pp-step-mset* :: (*f*,*v*,*s*)*pat-problem-mset* \Rightarrow (*f*,*v*,*s*)*pats-problem-mset* \Rightarrow *bool*

(**infix** $\langle \Rightarrow_m \rangle$ 50) **where**
pat-remove-pp: *add-mset* {#} *pp* \Rightarrow_m {#}
| *pat-simp-mp*: *mp-step-mset mp mp'* \implies *add-mset mp pp* \Rightarrow_m {#} (*add-mset mp'* *pp*) {#}
| *pat-remove-mp*: *match-fail mp* \implies *add-mset mp pp* \Rightarrow_m {#} *pp* {#}
| *pat-instantiate*: *tvars-disj-pp* {*n* ..< *n+m*} (*pat-mset* (*add-mset mp pp*)) \implies
(*Var x*, *l*) \in *mp-mset mp* \wedge *is-Fun l* \vee
 \neg *improved* \wedge (*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*)
 \implies
add-mset mp pp \Rightarrow_m *mset* (*map* ($\lambda \tau$. *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* \implies *pp* \neq {#}
 \implies *Ball* (*tvars-pat* (*pat-mset pp'*)) (λx . \neg *inf-sort* (*snd x*)) \implies
(\neg *improved* \implies *pp'* = {#})
 \implies *pp* + *pp'* \Rightarrow_m {#} *pp'* {#}

inductive-set *pp-nd-step-mset* :: (*f*,*v*,*s*)*pat-problem-mset* *rel* ($\langle \Rightarrow_{nd} \rangle$) **where**
pp \Rightarrow_m *P* \implies *p'* \in # *P* \implies (*pp*, *p'*) \in \Rightarrow_{nd}

inductive *P-step-mset* :: (*f*,*v*,*s*)*pats-problem-mset* \Rightarrow (*f*,*v*,*s*)*pats-problem-mset* \Rightarrow *bool*

(**infix** $\langle \Rightarrow_m \rangle$ 50) **where**
P-failure: *add-mset* {#} *P* \neq *bottom-mset* \implies *add-mset* {#} *P* \Rightarrow_m *bottom-mset*
| *P-simp-pp*: *pp* \Rightarrow_m *pp'* \implies *add-mset pp P* \Rightarrow_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* ($\langle \Rightarrow \rangle$) **where**
 \Rightarrow = {(*P*, *P'*). *P* \Rightarrow_m *P'*}

6.2 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*:
 $\exists n. \text{tvars-disj-pp } \{n..<n + m\} (\text{pat-mset } p)$
proof –
define p' **where** $p' = \text{pat-mset } p$
define V **where** $V = \text{fst } ' \cup (\text{vars } ' (\text{fst } ' \cup p'))$
have *finite* V **unfolding** $V\text{-def}$ $p'\text{-def}$ **by** *auto*
define n **where** $n = \text{Suc } (\text{Max } V)$
{
 fix mp t l
 assume $mp \in p' (t,l) \in mp$
 hence $\text{sub: } \text{fst } ' \text{ vars } t \subseteq V$ **unfolding** $V\text{-def}$ **by** *force*
 {
 fix x
 assume $x \in \text{fst } ' \text{ vars } t$
 with sub **have** $x \in V$ **by** *auto*
 with $\langle \text{finite } V \rangle$ **have** $x \leq \text{Max } V$ **by** *simp*
 also **have** $\dots < n$ **unfolding** $n\text{-def}$ **by** *simp*
 finally **have** $x < n$.
 }
 hence $\text{fst } ' \text{ vars } t \cap \{n..<n + m\} = \{\}$ **by** *force*
}
}
thus *?thesis* **unfolding** tvars-disj-pp-def $p'\text{-def}$ [*symmetric*]
by (*intro exI[of - n] ballI, force*)
qed

lemma *mp-mset-in-pat-mset*: $mp \in \# pp \implies mp\text{-mset } mp \in \text{pat-mset } pp$
by *auto*

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

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

end

lemma *count-le-size*: $count\ A\ x \leq size\ A$
by (*induct A, auto*)

lemma *Max-le-MaxI*: **assumes** $finite\ A\ A \neq \{\}$ $finite\ B$
 $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$
shows $Max\ A \leq Max\ B$
using *assms* **by** (*metis Max-ge-iff Max-in empty-iff*)

lemma *steps-bound*: **assumes** $\bigwedge x\ y. (x,y) \in r \implies f\ x > f\ y$
and $(x,y) \in r \widetilde{n}$
shows $f\ x \geq f\ y + n$
using *assms*(2)
proof (*induct n arbitrary: x y*)
case (*Suc n x z*)
then **obtain** y **where** $(x,y) \in r \widetilde{n}$ **and** $(y,z) \in r$ **by** *auto*
from *Suc*(1)[*OF this*(1)] *assms*(1)[*OF this*(2)]
show ?case **by** *auto*
qed *auto*

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

```

lemma pat-empty-or-trans-or-finite-constr-form:
  fixes  $p :: ('f, 'v, 's) \text{ pat-problem-mset}$ 
  assumes inf: improved  $\implies$  infinite (UNIV :: 'v set) and wf: wf-pat (pat-mset
p)
  shows  $p = \{\#\} \vee (\exists ps. p \Rightarrow_m ps) \vee (\text{improved} \wedge \text{finite-constr-form-pat } C$ 
(pat-mset p))
proof (cases  $p = \{\#\}$ )
  case True
  thus ?thesis by auto
next
  case pne: False
  from pat-mset-fresh-vars obtain n where fresh: tvars-disj-pp  $\{n..<n + m\}$ 
(pat-mset p) by blast
  show ?thesis
  proof (cases  $\{\#\} \in\# p$ )
  case True
  then obtain  $p'$  where  $p = \text{add-mset } \{\#\} p'$  by (rule mset-add)
  with pat-remove-pp show ?thesis by auto
next
  case empty-p: False
  show ?thesis
  proof (cases  $\exists mp\ s\ t. mp \in\# p \wedge (s,t) \in\# mp \wedge \text{is-Fun } t$ )
  case True
  then obtain  $mp\ s\ t$  where  $mp: mp \in\# p$  and  $(s,t) \in\# mp$  and is-Fun  $t$  by
auto
  then obtain  $g\ ts$  where  $(s, \text{Fun } g\ ts) \in\# mp$  by (cases  $t$ , auto)
  from  $mp$  obtain  $p'$  where  $p: p = \text{add-mset } mp\ p'$  by (rule mset-add)
  from  $mem$  obtain  $mp'$  where  $mp: mp = \text{add-mset } (s, \text{Fun } g\ ts)\ mp'$  by (rule
mset-add)
  show ?thesis
  proof (cases  $s$ )
  case  $s: (\text{Fun } f\ ss)$ 
  from pat-simp-mp[OF match-decompose, of f ss] pat-remove-mp[OF match-clash,
of f ss]
  show ?thesis unfolding  $p\ mp\ s$  by blast
next
  case (Var  $x$ )
  from  $Var\ mem$  obtain  $l$  where  $(Var\ x, l) \in\# mp \wedge \text{is-Fun } l$  by auto
  from pat-instantiate[OF fresh[unfolded  $p$ ] disjI1[OF this]]
  show ?thesis unfolding  $p$  by auto
qed
next
  case False
  hence rhs-vars:  $\bigwedge mp\ s\ l. mp \in\# p \implies (s,l) \in\# mp \implies \text{is-Var } l$  by auto
  let ?single-var =  $(\exists mp\ t\ x. \text{add-mset } (t, Var\ x)\ mp \in\# p \wedge x \notin \bigcup (\text{vars } 's$ 
snd ' set-mset  $mp))$ 
  let ?duplicate =  $(\exists mp\ pair. \text{add-mset } pair\ (\text{add-mset } pair\ mp) \in\# p)$ 
  show ?thesis
  proof (cases  $?single-var \vee ?duplicate$ )

```

```

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 s x
      assume mpp: mp ∈# p and sx: (s, Var x) ∈# mp
      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) ∈# mp' and x ∈ vars (snd (t,l)) by force
      with rhs-vars[OF mpp, of t l] have tx: (t, Var x) ∈# mp' unfolding mp
    by auto
      then obtain mp'' where mp': mp' = add-mset (t, Var x) mp'' by (rule
      mset-add)
      from ndup[simplified, rule-format] mpp have s ≠ t unfolding mp mp' by
      auto
      hence  $\exists t mp'. mp = add-mset (s, Var x) (add-mset (t, Var x) mp') \wedge s$ 
       $\neq t$  unfolding mp mp' by auto
    } note twoX = this
    {
      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] obtain x where sx: (s, Var x) ∈# mp by
      (cases l, auto)
      from twoX[OF mpp sx]
      have  $\exists s t x mp'. mp = add-mset (s, Var x) (add-mset (t, Var x) mp') \wedge$ 
       $s \neq t$  by blast
    }

```

```

} note two = this
show ?thesis
proof (cases  $\exists mp\ s\ t\ x. add\_mset\ (s, Var\ x)\ (add\_mset\ (t, Var\ x)\ mp) \in\#$ 
 $p \wedge Conflict\_Clash\ s\ t$ )
  case True
  then obtain  $mp\ s\ t\ x$  where
     $mp: add\_mset\ (s, Var\ x)\ (add\_mset\ (t, Var\ x)\ mp) \in\#\ p$  (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 improved)
  case not-impr: False
  show ?thesis
  proof (cases  $\exists mp\ s\ t\ x\ y. add\_mset\ (s, Var\ x)\ (add\_mset\ (t, Var\ x)\ mp)$ 
 $\in\# p \wedge Conflict\_Var\ s\ t\ y \wedge \neg inf\_sort\ (snd\ y)$ )
    case True
    from True obtain  $mp\ s\ t\ x\ y$  where
       $mp: add\_mset\ (s, Var\ x)\ (add\_mset\ (t, Var\ x)\ mp) \in\# p$  (is ?mp  $\in\#$ 
- ) and conf: Conflict-Var s t y and  $y: \neg inf\_sort\ (snd\ y)$ 
    by blast
    from mp obtain  $p'$  where  $p: p = add\_mset\ ?mp\ p'$  by (rule mset-add)
    let ?mp =  $add\_mset\ (s, Var\ x)\ (add\_mset\ (t, Var\ x)\ mp)$ 
    from pat-instantiate[OF - disjI2, of n ?mp p' s t y, folded p, OF fresh]
    show ?thesis using y conf not-impr by auto
  next
  case no-non-inf: False
  have  $\exists ps. p + \{\#\} \Rightarrow_m ps$ 
  proof (intro exI, rule pat-inf-var-conflict[OF - pne], intro ballI)
    fix mp
    assume  $mp: mp \in pat\_mset\ p$ 
    then obtain  $mp'$  where  $mp': mp' \in\# p$  and  $mp: mp = mp\_mset$ 
 $mp'$  by auto
    from two[OF mp']
    obtain  $s\ t\ x\ mp''$ 
    where  $mp'': mp' = add\_mset\ (s, Var\ x)\ (add\_mset\ (t, Var\ x)\ mp'')$ 
and diff:  $s \neq t$  by auto
    from conflicts(3)[OF diff] obtain y where Conflict-Clash s t  $\vee$ 
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
  next
    case impr: True
    define exVar where exVar mp = ( $\forall x t. (t, \text{Var } x) \in\# mp \longrightarrow (\exists y. (\text{Var } y, \text{Var } x) \in\# mp)$ )
    for mp :: ('f,'v,'s)match-problem-mset
    show ?thesis
    proof (cases  $\forall mp \in\# p. exVar mp$ )
      case False
      then obtain mp where mpp:  $mp \in\# p$  and  $\neg exVar mp$  by auto
      from this[unfolded exVar-def] obtain s x where sx:  $(s, \text{Var } x) \in\# mp$ 
and
      no-var:  $\bigwedge y. (\text{Var } y, \text{Var } x) \notin\# mp$ 
      by auto
      from no-var have allFun:  $(t, \text{Var } x) \in\# mp \implies is-Fun t$  for t by
(cases t, auto)
      from sx no-var obtain f ss where s:  $s = Fun f ss$  by (cases s, auto)
      from twoX[OF mpp sx] obtain t mp' where stx:  $mp = add-mset (s, \text{Var } x) (add-mset (t, \text{Var } x) mp')$ 
      and st:  $s \neq t$  by auto
      let ?Var = Var :: 'v  $\Rightarrow$  ('f, 'v)term
      let ?f =  $\lambda tl. snd tl = ?Var x$ 
      define mp1 where mp1 = filter-mset ?f mp
      have size:  $size mp1 \geq 2$  unfolding mp1-def stx by auto
      define mp2 where mp2 = filter-mset (Not o ?f) mp
      {
        fix t l
        assume (t,l)  $\in\# mp2$ 
        hence (t,l)  $\in\# mp$  and  $l \neq \text{Var } x$  unfolding mp2-def by auto
        from rhs-vars[OF mpp this(1)] this(2) have  $x \notin vars l$  by (cases l,
auto)
      }
      note mp2 = this
      define n where n = length ss
      with s have rtS:  $root s = Some (f,n)$  unfolding n-def by auto
      from stx have smp:  $(s, \text{Var } x) \in\# mp$  by auto
      {
        fix t l
        assume (t,l)  $\in\# mp1$ 
        from this[unfolded mp1-def]
        have l:  $l = \text{Var } x$  and tmp:  $(t, \text{Var } x) \in\# mp$  by auto
        from allFun tmp obtain g ts where t:  $t = Fun g ts$  by (cases t, auto)
        {
          assume rtT:  $root t \neq Some (f,n)$ 
          hence st:  $s \neq t$  using rtS by auto
          from rtS rtT have clash: Conflict-Clash s t unfolding s t
          by (auto simp: conflicts.simps)
          from smp tmp st have  $\exists mp'. mp = add-mset (s, \text{Var } x) (add-mset$ 
(t, Var x) mp')

```

```

    by (metis insert-noteq-member multi-member-split prod.inject)
  with clash no-clash mpp have False by blast
}
hence  $l = \text{Var } x \wedge \text{root } t = \text{Some } (f,n)$  using  $l$  by auto
} note  $mp1 = \text{this}$ 

define  $VV$  where  $VV = \bigcup (\text{vars } ' \text{snd } ' \text{mp-mset } mp)$ 
have finite  $VV$  by (auto simp:  $VV$ -def)
with inf[ $OF$   $impr$ ] have infinite  $(UNIV - VV)$  by auto
then obtain  $Ys$  where  $Ys: Ys \subseteq UNIV - VV$  card  $Ys = n$  finite  $Ys$ 
  by (meson infinite-arbitrarily-large)
  from  $Ys(2-3)$  obtain  $ys$  where  $ys: \text{distinct } ys$  length  $ys = n$  set  $ys$ 
=  $Ys$ 
  by (metis distinct-card finite-distinct-list)
with  $Ys$  have  $dist: VV \cap \text{set } ys = \{\}$  by auto
have  $disj: \text{lvars-disj-mp } ys$  ( $mp$ -mset  $mp$ ) length  $ys = n$ 
  unfolding  $\text{lvars-disj-mp-def}$  using  $ys$   $dist$  unfolding  $VV$ -def by auto
have  $mp = mp1 + mp2$  unfolding  $mp1$ -def  $mp2$ -def by simp
  from  $\text{match-decompose}'[of mp1 x - - mp2, \text{folded } \text{this}, OF mp1 mp2$ 
 $disj$  size  $impr]$ 
  obtain  $mp'$  where  $mp \rightarrow_m mp'$  by fast
  from  $\text{pat-simp-mp}[OF \text{this}] mpp$ 
  show ?thesis by (metis mset-add)
next
case  $exVar: True$ 
show ?thesis
proof (cases  $\forall mp \in \# p. (\forall t l. (t,l) \in \# mp \longrightarrow \text{is-Var } l \wedge \mathcal{T}(C,\mathcal{V})$ 
 $t \neq \text{None}))$ )
  case False
  then obtain  $mp$   $s$   $l$  where  $mpp: mp \in \# p$  and  $sl: (s,l) \in \# mp$  and
     $ch: \neg \text{is-Var } l \vee \mathcal{T}(C,\mathcal{V})$   $s = \text{None}$ 
    by auto
  from  $\text{rhs-vars}[OF mpp sl]$  obtain  $x$  where  $l: l = \text{Var } x$  by auto
  with  $ch$  have  $\text{None}: \mathcal{T}(C,\mathcal{V})$   $s = \text{None}$  by auto
  from  $\text{None}$  obtain  $f$   $ss$  where  $s: s = \text{Fun } f$   $ss$  by (cases  $s$ , auto)
  from  $sl$   $l$  have  $sx: (s, \text{Var } x) \in \# mp$  by auto
  from  $exVar[\text{unfolded } exVar\text{-def}, \text{rule-format}, OF mpp sx]$  obtain  $y$ 
    where  $(\text{Var } y, \text{Var } x) \in \# mp$  by auto
  with  $sx$  obtain  $mp'$  where  $mp = \text{add-mset } (\text{Var } y, \text{Var } x)$  ( $\text{add-mset}$ 
 $(s, \text{Var } x)$   $mp'$ )
  unfolding  $s$  by (metis insert-noteq-member is-FunI is-VarI mset-add
 $\text{prod.inject}$ )
  from  $\text{match-clash-sort}[of \text{Var } y s x mp', \text{unfolded } \text{None}, \text{folded } \text{this}]$ 
  have  $\text{match-fail } mp$  by auto
  from  $\text{pat-remove-mp}[OF \text{this}] mpp$ 
  show ?thesis by (metis mset-add)
next
case  $constr\text{-form}: True$ 
  define  $finmp$  where  $finmp$   $mp = (\forall t l. (t, l) \in \# mp \longrightarrow (\exists \iota.$ 

```

```

finite-sort C  $\iota \wedge t : \iota$  in  $\mathcal{T}(C, \mathcal{V})$ )
  for mp :: ('f, 'v, 's)match-problem-mset
  show ?thesis
  proof (cases  $\forall mp \in\# p. \text{finmp } mp$ )
  case True
  {
    fix mp
    assume mp: mp  $\in\# p$ 
    hence finmp mp using True by auto
    with constr-form mp have finite-constr-form-mp C (mp-mset mp)
      unfolding finmp-def by (simp add: finite-constr-form-mp-def)
    }
  thus ?thesis using impr unfolding finite-constr-form-pat-def by
auto

next
case someInf: False
show ?thesis
proof (cases  $\exists s t x mp. mp \in\# p \wedge (s, \text{Var } x) \in\# mp \wedge (t, \text{Var } x) \in\# mp \wedge \mathcal{T}(C, \mathcal{V}) s \neq \mathcal{T}(C, \mathcal{V}) t$ )
case True
then obtain s t x mp where mp: mp  $\in\# p$  and s: (s, Var x)  $\in\# mp$  and t: (t, Var x)  $\in\# mp$ 
  and sort-clash:  $\mathcal{T}(C, \mathcal{V}) s \neq \mathcal{T}(C, \mathcal{V}) t$ 
  by auto
  from sort-clash have st: s  $\neq t$  by auto
  with s t obtain mp' where mp = add-mset (s, Var x) (add-mset (t, Var x) mp')
    by (metis insert-noteq-member mset-add prod.inject)
  from match-clash-sort[of s t x mp', folded this] sort-clash
  have match-fail mp by auto
  from pat-remove-mp[OF this] show ?thesis using mp
    by (metis mset-add)
next
case False
hence noSortClash:  $\bigwedge s t x mp. mp \in\# p \implies (s, \text{Var } x) \in\# mp \implies (t, \text{Var } x) \in\# mp \implies \mathcal{T}(C, \mathcal{V}) s = \mathcal{T}(C, \mathcal{V}) t$ 
  by blast

define p1 where p1 = filter-mset (Not o finmp) p
define p2 where p2 = filter-mset finmp p
have p: p = p1 + p2 unfolding p1-def p2-def by simp
have p  $\Rightarrow_m \{\#p2\#$  unfolding p
proof (rule pat-inf-var-conflict[of p1 p2]; (intro ballI, clarsimp)?)
{
  from someInf obtain mp where mp  $\in\# p$  and  $\neg \text{finmp } mp$ 

  hence mp  $\in\# p1$  unfolding p1-def by auto
  thus p1  $\neq \{\#$  by auto
}
}

```

```

      {
        fix mp
        assume mp ∈# p1
        from this[unfolded p1-def] have nfin: ¬ finmp mp and mp: mp
∈# p by auto
        from nfin[unfolded finmp-def, simplified]
        obtain t l where tl: (t, l) ∈# mp and inf: ∧ ι. finite-sort C
ι ⇒ ¬ t : ι in T(C, V)
        by auto
        from constr-form[rule-format, OF mp tl] have l: is-Var l and
sorted: T(C, V) t ≠ None
        by auto
        from l obtain x where l: l = Var x by auto
        from sorted obtain ι where sorted: t : ι in T(C, V) by (cases
T(C, V) t, auto simp: hastype-def)
        from inf sorted have inf: ¬ finite-sort C ι by auto
        from tl l have tx: (t, Var x) ∈# mp by auto
        from exVar[unfolded exVar-def, rule-format, OF mp tx] obtain
y
          where yx: (Var y, Var x) ∈# mp by auto
          have y: Var y : snd y in T(C, V) by simp
          from noSortClash[OF mp yx tx] sorted y inf
          have inf: ¬ finite-sort C (snd y) by (auto simp: hastype-def)
          from wf[unfolded wf-pat-def wf-match-def tvars-match-def,
simplified, rule-format, OF mp]
          yx
          have snd y ∈ S by force
          with inf have inf: inf-sort (snd y) using inf-sort by auto
          from twoX[OF mp yx]
          obtain t mp' where mp': mp = add-mset (Var y, Var x)
(add-mset (t, Var x) mp')
          and yt: Var y ≠ t by auto
          from mp' have tx: (t, Var x) ∈# mp by auto
          from noSortClash[OF mp yx tx] y
          have t: t : snd y in T(C, V) by (auto simp: hastype-def)
          obtain cs where conf: conflicts (Var y) t = Some cs y ∈ set
cs
            using t yt by (cases t, auto simp: conflicts.simps)
            show inf-var-conflict (mp-mset mp)
            unfolding inf-var-conflict-def
            by (intro exI conjI, rule yx, rule tx, insert inf conf, auto)
      }
    {
      fix x
      assume x: x ∈ tvars-pat (mp-mset ' set-mset p2) and inf:
inf-sort (snd x)
      from x[unfolded tvars-pat-def tvars-match-def]
      obtain mp t l where mp: mp ∈# p2 and tl: (t, l) ∈# mp and
x: x ∈ vars t by auto

```

```

    from mp[unfolded p2-def] have fin: finmp mp and mp: mp ∈#
p by auto
    from wf[unfolded wf-pat-def wf-match-def, simplified, rule-format,
OF mp] x tl
    have xS: snd x ∈ S unfolding tvars-match-def by auto
    from inf-sort[OF this] inf have inf: ¬ finite-sort C (snd x)
by auto
    from constr-form[rule-format, OF mp tl] obtain y where l: l
= Var y
    and sorted:  $\mathcal{T}(C, \mathcal{V})$  t ≠ None by auto
    note ty = tl[unfolded l]
    from fin[unfolded finmp-def, rule-format, OF tl] obtain  $\iota$ 
where
    fin: finite-sort C  $\iota$  and sorted: t :  $\iota$  in  $\mathcal{T}(C, \mathcal{V})$  by auto
    from sorted x fin have finite-sort C (snd x)
    proof (induct)
    case (Fun f ss  $\sigma$  s  $\tau$ )
    then obtain s where s: s ∈ set ss and x: x ∈ vars s by auto
    from s obtain i where i: i < length ss and si: s = ss ! i
by (auto simp: set-conv-nth)
    from Fun(2) si i have s :  $\sigma$  s ! i in  $\mathcal{T}(C, \mathcal{V})$ 
    and i < length  $\sigma$  s unfolding list-all2-conv-all-nth by auto
    hence  $\sigma$  s ! i ∈ set  $\sigma$  s by auto
    from finite-arg-sort[OF Fun(5,1) this] have finite-sort C ( $\sigma$  s
! i) by auto
    with Fun(3) x show finite-sort C (snd x) unfolding si
using i
    unfolding list-all2-conv-all-nth by auto
    qed auto
    with inf have False by simp
}
    thus  $\bigwedge x \iota. (x, \iota) \in \text{tvvars-pat} (\text{mp-mset } \text{' set-mset } p2) \implies$ 
inf-sort  $\iota \implies \text{False}$ 
    by auto
    qed (insert impr, auto)
    thus ?thesis by auto
qed
qed
qed
qed
qed
qed
qed
qed
qed

```

context

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

lemma *pat-empty-or-trans*: *wf-pat* (*pat-mset* *p*) $\implies p = \{\#\} \vee (\exists ps. p \Rightarrow_m ps)$
using *pat-empty-or-trans-or-finite-constr-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*: $P \in NF \Rightarrow$
shows $P \in \{\{\#\}, \textit{bottom-mset}\}$
proof (*rule ccontr*)
assume *nNF*: $P \notin \{\{\#\}, \textit{bottom-mset}\}$
from *NF* **have** *NF*: $\neg (\exists Q. P \Rightarrow_m Q)$ **unfolding** *P-step-def* **by** *blast*
from *nNF* **obtain** *p P'* **where** $P = \textit{add-mset } p P'$
using *multiset-cases* **by** *auto*
with *wf* **have** *wf-pat* (*pat-mset* *p*) **by** (*auto simp: wf-pats-def*)
with *pat-empty-or-trans*
obtain *ps* **where** $p = \{\#\} \vee p \Rightarrow_m ps$ **by** *auto*
with *P-simp-pp*[*of p ps*] *NF*
have $p = \{\#\}$ **unfolding** *P* **by** *auto*
from *P-failure*[*of P'*] *P this nNF NF* **show** *False* **by** *blast*
qed
end

context

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

lemma *pat-empty-or-trans-or-fvf*:

fixes *p* :: ('*f*, '*v*, '*s*) *pat-problem-mset*
assumes *wf-pat* (*pat-mset* *p*)
shows $p = \{\#\} \vee (\exists ps. p \Rightarrow_m ps) \vee \textit{finite-constr-form-pat } C (\textit{pat-mset } p)$
using *assms pat-empty-or-trans-or-finite-constr-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*: $(P::('f, 'v, 's) \textit{pat-problem-mset}) \in NF \Rightarrow$
and *p*: $p \in \# P$
shows *finite-constr-form-pat* *C* (*pat-mset* *p*)
proof (*rule ccontr*)
assume *nfvf*: $\neg ?thesis$
from *wf p* **have** *wfp*: *wf-pat* (*pat-mset* *p*) **by** (*auto simp: wf-pats-def*)
from *mset-add*[*OF p*] **obtain** *P'* **where** $P = \textit{add-mset } p P'$ **by** *auto*
from *NF* **have** *NF*: $\neg (\exists Q. P \Rightarrow_m Q)$ **unfolding** *P-step-def* **by** *blast*
from *pat-empty-or-trans-or-fvf*[*OF wfp*] *nfvf*

obtain ps **where** $p = \{\#\} \vee p \Rightarrow_m ps$ **by** *auto*
with P -*simp-pp*[*of p ps*] NF
have $p = \{\#\}$ **unfolding** P **by** *auto*
with *nfvf* **show** *False* **unfolding** *finite-constr-form-pat-def* **by** *auto*
qed

lemma *pp-step-mset-empty-cong*: **assumes** *improved*
shows $p \Rightarrow_m P \implies p + \text{replicate-mset } n \ \{\#\} \Rightarrow_m \text{image-mset } (\lambda p'. p' + \text{replicate-mset } n \ \{\#\}) P$
proof (*induct rule: pp-step-mset.induct*)
case (*pat-remove-pp pp*)
show *?case* **by** *simp* (*rule pat-remove-pp*)
next
case *: (*pat-simp-mp mp mp' pp*)
show *?case* **using** *pat-simp-mp*[*OF **] **by** *auto*
next
case *: (*pat-remove-mp mp pp*)
show *?case* **using** *pat-remove-mp*[*OF **] **by** *auto*
next
case *: (*pat-instantiate n' mp pp x l s y t*)
define τ **where** $\tau = \text{mset } (\tau\text{-list } n' x)$
define p **where** $p = \text{add-mset } mp \ pp$
from *(1) **have** $\text{tvars-disj-pp } \{n'..<n' + m\} (\text{pat-mset } (\text{add-mset } mp \ (pp + \text{replicate-mset } n \ \{\#\})))$
by (*fastforce simp: tvars-disj-pp-def*)
from *pat-instantiate*[*OF this *(2)*]
have $\text{add-mset } mp \ (pp + \text{replicate-mset } n \ \{\#\}) \Rightarrow_m \text{mset } (\text{map } (\lambda\tau. \text{subst-pat-problem-mset } \tau \ (p + \text{replicate-mset } n \ \{\#\})) \ (\tau\text{-list } n' x))$
by (*simp add: p-def*)
also have $\text{mset } (\text{map } (\lambda\tau. \text{subst-pat-problem-mset } \tau \ (p + \text{replicate-mset } n \ \{\#\})) \ (\tau\text{-list } n' x))$
 $= \{\#p' + \text{replicate-mset } n \ \{\#\}$
 $\ . p' \in \# \text{mset } (\text{map } (\lambda\tau. \text{subst-pat-problem-mset } \tau \ p) \ (\tau\text{-list } n' x))\#\}$
unfolding *tau-def*[*symmetric*] *subst-pat-problem-mset-def* *subst-match-problem-mset-def*
mset-map
image-mset.compositionality *o-def* **by** *auto*
finally show *?case* **by** (*auto simp: p-def*)
next
case *: (*pat-inf-var-conflict pp pp'*)
have $pp + (pp' + \text{replicate-mset } n \ \{\#\}) \Rightarrow_m \{\#pp' + \text{replicate-mset } n \ \{\#\}\#\}$
by (*rule pat-inf-var-conflict*[*OF *(1-2)*],
*insert *(3) <improved>*, *auto simp: tvars-pat-def tvars-match-def*)
thus *?case* **by** (*auto simp: ac-simps*)
qed

theorem *nd-step-NF-fvf*: **fixes** $p :: (f, 'v, 's) \text{pat-problem-mset}$
assumes *wf-pat* (*pat-mset p*)
and $p \in NF \Rightarrow_{nd}$

```

shows finite-constr-form-pat C (pat-mset p)
proof -
  define p1 where p1 = filter-mset ((=) {#}) p
  define p2 where p2 = filter-mset ((≠) {#}) p
  have p: p = p1 + p2 by (auto simp: p1-def p2-def)
  from assms(1) have wf: wf-pat (pat-mset p2) unfolding p2-def wf-pat-def by
  auto
  define n where n = size p1
  have p1: p1 = replicate-mset n {#} unfolding n-def p1-def
    by (metis (mono-tags, lifting) count-conv-size-mset filter-eq-replicate-mset fil-
  ter-mset-cong0)
  with p have p: p = p2 + replicate-mset n {#} by simp
  {
    fix q
    assume (p2,q) ∈ ⇒nd
    then obtain Q where p2 ⇒m Q q ∈# Q by cases
    from pp-nd-step-mset.intros[OF pp-step-mset-empty-cong[OF ‹improved› this(1),
  of n, folded p]]
      this(2)
    have ∃ q'. (p,q') ∈ ⇒nd by auto
    with assms have False by auto
  }
  hence NF: p2 ∈ NF ⇒nd by auto
  from pat-empty-or-trans-or-fwf[OF wf]
  have finite-constr-form-pat C (pat-mset p2)
  proof (elim disjE)
    show p2 = {#} ⇒ ?thesis by (auto simp: finite-constr-form-pat-def)
    assume ∃ P. p2 ⇒m P
    then obtain P where step: p2 ⇒m P by auto
    with NF have P: P = {#}
    by (meson NF-no-step multiset-nonemptyE pattern-completeness-context.pp-nd-step-mset.simps)
    from step[unfolded P]
    show ?thesis
    proof (cases)
      case (pat-remove-pp pp)
      hence {#} ∈# p2 by auto
      from this[unfolded p2-def] have False by simp
      then show ?thesis by auto
    next
      case *: (pat-instantiate n mp pp x l s y t)
      from * have set (τs-list n x) = {} by auto
      from this[unfolded τs-list-def] have set (Cl (snd x)) = {} by auto
      from this[unfolded Cl] have empty: {(f, ss). f : ss → snd x in C} = {} by
  auto
      from * ‹improved› have (Var x, l) ∈# mp by auto
      with *(1) have x ∈ tvars-pat (pat-mset p2)
      by (force simp: tvars-pat-def tvars-match-def)
      with wf have snd x ∈ S unfolding wf-pat-def wf-match-def tvars-pat-def by
  force

```

```

    hence  $\neg$  empty-sort  $C$  (snd  $x$ ) by (rule not-empty-sort)
    then obtain  $t$  where  $t : \text{snd } x$  in  $\mathcal{T}(C)$  unfolding empty-sort-def by auto
    from this empty have False by (induct, auto)
    then show ?thesis by auto
  qed auto
qed auto
thus ?thesis unfolding  $p$  finite-constr-form-pat-def
  by (auto simp: finite-constr-form-mp-def)
qed
end
end

```

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

```

```

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

```

```

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

```

```

lemma fun-diff-num-funs: fun-diff  $l$   $t \leq$  num-funs  $l$ 

```

```

proof (induct  $l$   $t$  rule: fun-diff.induct)

```

```

  case (2  $f$   $ls$   $g$   $ts$ )

```

```

  show ?case

```

```

  proof (cases  $f = g \wedge \text{length } ts = \text{length } ls$ )

```

```

    case True

```

```

      have sum-list (map2 fun-diff  $ls$   $ts$ )  $\leq$  sum-list (map num-funs  $ls$ )

```

```

        by (intro sum-list-mono2, insert True 2, (force simp: set-zip)+)

```

```

      with 2 show ?thesis by auto

```

```

    qed auto

```

```

  qed auto

```

```

lemma fun-diff-subst: fun-diff  $l$  ( $t \cdot \sigma$ )  $\leq$  fun-diff  $l$   $t$ 

```

```

proof (induct  $l$  arbitrary:  $t$ )

```

```

  case  $l$ : (Fun  $f$   $ls$ )

```

```

  show ?case

```

```

  proof (cases  $t$ )

```

```

    case  $t$ : (Fun  $g$   $ts$ )

```

```

      show ?thesis unfolding  $t$  using  $l$  by (auto intro: sum-list-mono2)

```

```

next
  case t: (Var x)
  show ?thesis unfolding t using fun-diff-num-funs[of Fun f ls] by auto
qed
qed auto

```

```

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

```

```

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

```

```

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

```

```

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

```

```

fun syms-term :: ('f, 'v)term ⇒ ('v + 'f)multiset where
  syms-term (Var x) = {# Inl x #}
| syms-term (Fun f ts) = add-mset (Inr f) (sum-mset (image-mset syms-term (mset
ts)))

```

```

lemma vars-term-syms-term: x ∈ vars-term t ⇔ Inl x ∈# syms-term t
  by (induct t, auto)

```

lemma *replicate-mset-add*: $\text{replicate-mset } (n + m) a = \text{replicate-mset } n a + \text{replicate-mset } m a$

by (*metis repeat-mset-distrib repeat-mset-replicate-mset*)

lemma *syms-term-subst*: $\text{syms-term } (t \cdot \text{subst } x s) + \text{replicate-mset } (\text{count } (\text{syms-term } t) (\text{Inl } x)) (\text{Inl } x)$

$= \text{syms-term } t + \text{repeat-mset } (\text{count } (\text{syms-term } t) (\text{Inl } x)) (\text{syms-term } s)$ (**is** $?l$ $t = ?r t$)

proof (*induct t*)

case (*Var y*)

show $?case$ **unfolding** *subst-def* **by** *auto*

next

case (*Fun f ts*)

have $?case \longleftrightarrow$

$(\sum \# (\text{image-mset } \text{syms-term } \{\#sa \cdot \text{subst } x s. sa \in \# \text{mset } ts\# \}) + \text{replicate-mset } (\text{count } (\sum \# (\text{image-mset } \text{syms-term } (\text{mset } ts))) (\text{Inl } x)) (\text{Inl } x))$

$=$

$\sum \# (\text{image-mset } \text{syms-term } (\text{mset } ts)) + \text{repeat-mset } (\text{count } (\sum \# (\text{image-mset } \text{syms-term } (\text{mset } ts))) (\text{Inl } x)) (\text{syms-term } s)$

$s))$

(**is** $- \longleftrightarrow ?ls ts = ?rs ts$) **by** *simp*

also have \dots **using** *Fun*

proof (*induct ts*)

case (*Cons t ts*)

have $?rs (t \# ts) = ?r t + ?rs ts$ **by** *auto*

also have $\dots = ?l t + ?ls ts$ **using** *Cons* **by** *auto*

also have $\dots = ?ls (t \# ts)$ **by** (*simp add: replicate-mset-add*)

finally show $?case ..$

qed *auto*

finally show $?case$ **by** *simp*

qed

definition *num-syms* :: $(f, v) \text{term} \Rightarrow \text{nat}$ **where**

$\text{num-syms } t = \text{size } (\text{syms-term } t)$

lemma *num-syms-pos*[*simp*]: $\text{num-syms } t > 0$

unfolding *num-syms-def* **by** (*cases t, auto*)

lemma *num-syms-0*[*simp*]: $\text{num-syms } t \neq 0$

unfolding *num-syms-def* **by** (*cases t, auto*)

lemma *num-syms-subst*: $\text{num-syms } (t \cdot \text{subst } x s) = \text{num-syms } t + \text{count } (\text{syms-term } t) (\text{Inl } x) * (\text{num-syms } s - 1)$

proof $-$

let $?cx = \text{count } (\text{syms-term } t) (\text{Inl } x)$

from *arg-cong[OF syms-term-subst[of t x s], of size]*

have $\text{num-syms } (t \cdot \text{subst } x s) + ?cx = \text{num-syms } t + ?cx * \text{num-syms } s$

unfolding *size-union num-syms-def* **by** *simp*

from *arg-cong*[*OF this, of $\lambda n . n - ?cx$*]
have *num-syms* ($t \cdot \text{subst } x \ s$) = *num-syms* $t + ?cx * \text{num-syms } s - ?cx$ **by**
auto
also have $\dots = \text{num-syms } t + ?cx * (\text{num-syms } s - 1)$
using *num-syms-pos*[*of s*] **by** (*cases num-syms s, auto*)
finally show *?thesis* .
qed

lemma *num-syms-Fun*[*simp*]: *num-syms* (*Fun* $f \ ts$) = *Suc* (*sum-list* (*map num-syms*
 ts))
unfolding *num-syms-def*
by (*simp, induct ts, auto*)

abbreviation (*input*) *sum-ms* :: ($'a \Rightarrow 'b :: \text{comm-monoid-add}$) $\Rightarrow 'a \text{ multiset} \Rightarrow$
 $'b$ **where**
 $\text{sum-ms } f \ ms \equiv \text{sum-mset } (\text{image-mset } f \ ms)$

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

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

lemma τs -*list*: $\text{set } (\tau s\text{-list } n \ x) = \tau s \ n \ x$
unfolding τs -*list-def* τs -*def* **using** *Cl* **by** *auto*

lemma *num-syms- τc* : $\text{num-syms } (t \cdot \tau c \ n \ x \ (f, \sigma s)) = \text{num-syms } t + \text{count } (\text{syms-term } t)$
 $(\text{Inl } x) * \text{length } \sigma s$
unfolding *num-syms-subst* τc -*def* *split*
by (*simp add: num-syms-def image-mset.compositionality o-def*)

lemma *num-syms- τs* : **assumes** $\tau \in \tau s \ n \ x$
shows $\text{num-syms } (t \cdot \tau) \leq \text{num-syms } t + \text{count } (\text{syms-term } t) (\text{Inl } x) * m$
proof –
from *assms*[*unfolded* τs -*def*]
obtain $f \ ss$ **where** $\tau : \tau = \tau c \ n \ x \ (f, \ ss)$ **and** $f : ss \rightarrow \text{snd } x \ \text{in } C$ **by** *auto*
from m [*OF this*(2)] **have** $\text{len} : \text{length } ss \leq m$.
hence $\text{count } (\text{syms-term } t) (\text{Inl } x) * \text{length } ss \leq \text{count } (\text{syms-term } t) (\text{Inl } x) * m$
by *auto*
thus *?thesis* **unfolding** τ *num-syms- τc* **by** *auto*
qed

definition *meas-diff-mp* :: ($'f, 'v, 's$)*match-problem-mset* $\Rightarrow \text{nat}$ **where**
 $\text{meas-diff-mp} = \text{sum-ms } (\lambda (t, l). \text{fun-diff } l \ t)$

definition *meas-diff* :: ($'f, 'v, 's$)*pat-problem-mset* $\Rightarrow \text{nat}$ **where**
 $\text{meas-diff} = \text{sum-ms } \text{meas-diff-mp}$

definition *max-size* :: $'s \Rightarrow \text{nat}$ **where**

$max\text{-size } s = (if\ s \in S \wedge \neg\ inf\text{-sort } s\ then\ Maximum\ (size\ \{t.\ t : s\ in\ \mathcal{T}(C)\})\ else\ 0)$

definition $tsyms\text{-}mp :: (f, v, s)\text{match}\text{-}problem\text{-}mset \Rightarrow (nat \times 's + 'f)\ multiset$
where

$tsyms\text{-}mp\ mp = sum\text{-}ms\ (syms\text{-}term\ o\ fst)\ mp$

definition $num\text{-}tsyms\text{-}mp :: (f, v, s)\text{match}\text{-}problem\text{-}mset \Rightarrow nat$ **where**
 $num\text{-}tsyms\text{-}mp\ mp = sum\text{-}ms\ (num\text{-}syms\ o\ fst)\ mp$

definition $num\text{-}lsyms\text{-}mp :: (f, v, s)\text{match}\text{-}problem\text{-}mset \Rightarrow nat$ **where**
 $num\text{-}lsyms\text{-}mp\ mp = sum\text{-}ms\ (num\text{-}syms\ o\ snd)\ mp$

definition $num\text{-}syms\text{-}mp :: (f, v, s)\text{match}\text{-}problem\text{-}mset \Rightarrow nat$ **where**
 $num\text{-}syms\text{-}mp\ mp = num\text{-}tsyms\text{-}mp\ mp + num\text{-}lsyms\text{-}mp\ mp$

definition $num\text{-}syms\text{-}pat :: (f, v, s)\text{pat}\text{-}problem\text{-}mset \Rightarrow nat$ **where**
 $num\text{-}syms\text{-}pat = sum\text{-}ms\ num\text{-}syms\text{-}mp$

definition $meas\text{-}finvars\text{-}mp :: (f, v, s)\text{match}\text{-}problem\text{-}mset \Rightarrow nat$ **where**
 $meas\text{-}finvars\text{-}mp\ mp = sum\ (max\text{-}size\ o\ snd)\ (tvars\text{-}match\ (mp\text{-}mset\ mp))$

definition $max\text{-}dupl\text{-}mp$ **where**

$max\text{-}dupl\text{-}mp\ mp = Max\ (insert\ 0\ ((\lambda\ x.\ (\sum\ t \in \#image\text{-}mset\ fst\ mp.\ count\ (syms\text{-}term\ t)\ (Inl\ x)))\ \text{' } tvars\text{-}match\ (mp\text{-}mset\ mp)))$

lemma $max\text{-}dupl\text{-}mp\text{-}le\text{-}num\text{-}tsyms\text{-}mp$: $max\text{-}dupl\text{-}mp\ mp \leq num\text{-}tsyms\text{-}mp\ mp$

unfolding $max\text{-}dupl\text{-}mp\text{-}def$

proof (*subst Max-le-iff, force intro!: finite-imageI simp: tvars-match-def, force, intro ballI*)

fix n

assume $n: n \in insert\ 0\ \{\sum\ t \in \#image\text{-}mset\ fst\ mp.\ count\ (syms\text{-}term\ t)\ (Inl\ x)\}$
 $|. x \in tvars\text{-}match\ (mp\text{-}mset\ mp)$

show $n \leq num\text{-}tsyms\text{-}mp\ mp$

proof (*cases n = 0*)

case $False$

with n **obtain** x **where** $x: x \in tvars\text{-}match\ (mp\text{-}mset\ mp)$

and $n: n = (\sum\ t \in \#image\text{-}mset\ fst\ mp.\ count\ (syms\text{-}term\ t)\ (Inl\ x))$ **by** *auto*

have $nt: num\text{-}tsyms\text{-}mp\ mp = (\sum\ t \in \#image\text{-}mset\ fst\ mp.\ num\text{-}syms\ t)$

unfolding $num\text{-}tsyms\text{-}mp\text{-}def\ image\text{-}mset.compositionality ..$

show *?thesis* **unfolding** $n\ nt$

proof (*rule sum-mset-mono, goal-cases*)

case ($1\ t$)

show $count\ (syms\text{-}term\ t)\ (Inl\ x) \leq num\text{-}syms\ t$

unfolding $num\text{-}syms\text{-}def$ **by** (*rule count-le-size*)

qed

qed *auto*

qed

lemma *num-funs-le-num-syms*: $\text{num-funs } t \leq \text{num-syms } t$
by (*induct t*, *auto intro: sum-list-mono*)

lemma *fun-diff-le-num-syms*: $\text{fun-diff } l \ t \leq \text{num-syms } l$
proof (*induct l t rule: fun-diff.induct*)
case (*1 l x*)
then show *?case* **by** (*auto intro: num-funs-le-num-syms*)
next
case (*2 g ls f ts*)
show *?case*
proof (*cases f = g \wedge length ts = length ls*)
case *True*
have $\text{sum-list } (\text{map2 } \text{fun-diff } ls \ ts) \leq \text{sum-list } (\text{map2 } (\lambda \ l \ t. \ \text{num-syms } l) \ ls \ ts)$
by (*rule sum-list-mono, insert 2[OF True], auto*)
also have $\dots = \text{sum-list } (\text{map } \text{num-syms } ls)$
by (*rule arg-cong[of - - sum-list], insert True, auto intro!: nth-equalityI*)
finally show *?thesis* **by** *auto*
qed *auto*
qed *auto*

lemma *meas-diff-mp-le-num-lsyms-mp*: $\text{meas-diff-mp } mp \leq \text{num-lsyms-mp } mp$
unfolding *meas-diff-mp-def num-lsyms-mp-def o-def*
by (*rule sum-mset-mono, auto intro: fun-diff-le-num-syms*)

definition *meas-finvars* :: $(f, v, s)\text{pat-problem-mset} \Rightarrow \text{nat}$ **where**
meas-finvars = sum-ms meas-finvars-mp

definition *meas-tsymbols* :: $(f, v, s)\text{pat-problem-mset} \Rightarrow \text{nat}$ **where**
meas-tsymbols = sum-ms num-tsyms-mp

definition *meas-lsymbols* :: $(f, v, s)\text{pat-problem-mset} \Rightarrow \text{nat}$ **where**
meas-lsymbols = sum-ms num-lsyms-mp

definition *meas-dupl* :: $(f, v, s)\text{pat-problem-mset} \Rightarrow \text{nat}$ **where**
meas-dupl = sum-ms max-dupl-mp

lemma *tsyms-mp-num-tsyms*: $\text{num-tsyms-mp } mp = \text{size } (\text{tsyms-mp } mp)$
by (*induct mp, auto simp: num-tsyms-mp-def tsyms-mp-def num-syms-def*)

lemma *meas-dupl-le-num-syms-pat*: $\text{meas-dupl } p \leq \text{num-syms-pat } p$
unfolding *meas-dupl-def num-syms-pat-def num-syms-mp-def*
by (*rule sum-mset-mono, rule le-trans[OF max-dupl-mp-le-num-tsyms-mp], auto*)

lemma *meas-diff-le-num-syms-pat*: $\text{meas-diff } p \leq \text{num-syms-pat } p$
unfolding *meas-diff-def num-syms-pat-def num-syms-mp-def*
by (*rule sum-mset-mono, rule le-trans[OF meas-diff-mp-le-num-lsyms-mp], auto*)

lemma *tsyms-mp-subset-num-tsyms*: $\text{tsyms-mp } mp \subset\# \text{tsyms-mp } mp' \Longrightarrow \text{num-tsyms-mp}$

$mp < \text{num-tsyzms-mp } mp'$
unfolding *tsyzms-mp-num-tsyzms* **by** (rule *mset-subset-size*)

lemma *tsyzms-mp-mono*: **assumes** $mp \subseteq\# mp'$ **shows** $\text{tsyzms-mp } mp \subseteq\# \text{tsyzms-mp } mp'$

proof –
from *assms* **obtain** $mp2$ **where** $mp' = mp + mp2$
by (*metis subset-mset.less-eqE*)
thus *?thesis* **unfolding** *tsyzms-mp-def* **by** *auto*
qed

lemma *tsyzms-mp-strict-mono*: **assumes** $mp \subset\# mp'$ **shows** $\text{tsyzms-mp } mp \subset\# \text{tsyzms-mp } mp'$

proof –
from *subset-mset.lessE[OF assms]* **obtain** $mp1$ **where** $mp' = mp + mp1$ **and** $mp1 \neq \{\#\}$
by *blast*
then obtain tl $mp2$ **where** $mp' = \text{add-mset } tl (mp + mp2)$ **by** (*cases mp1, auto*)
thus *?thesis* **unfolding** *tsyzms-mp-def* **by** (*cases tl, cases fst tl; auto*)
qed

definition *measure-pat-poly* :: $\text{nat} \Rightarrow (f, 'v, 's)\text{pat-problem-mset} \Rightarrow \text{nat}$ **where**
 $\text{measure-pat-poly } c \ p = (c + \text{meas-diff } p) * (\text{meas-dupl } p * m + 1) + \text{meas-tsyzms}$
 p

lemma *measure-pat-poly*: $\text{measure-pat-poly } c \ p \leq (c + \text{num-syzms-pat } p) * (\text{num-syzms-pat } p * m + 2)$

proof –
have $\text{measure-pat-poly } c \ p \leq (c + \text{num-syzms-pat } p) * (\text{num-syzms-pat } p * m + 1) + \text{num-syzms-pat } p$
unfolding *measure-pat-poly-def*
proof (*intro add-mono mult-mono le-refl meas-dupl-le-num-syzms-pat meas-diff-le-num-syzms-pat*)
show $\text{meas-tsyzms } p \leq \text{num-syzms-pat } p$ **unfolding** *meas-tsyzms-def num-syzms-pat-def num-syzms-mp-def*
by (*intro sum-mset-mono, auto*)
qed *auto*
also have $\dots \leq (c + \text{num-syzms-pat } p) * (\text{num-syzms-pat } p * m + 2)$ **by** *simp*
finally show *?thesis* .
qed

lemma *measure-expr-decrease*: **assumes** $d1 < (d2 :: \text{nat})$ $du1 \leq du2$ $\text{sym1} \leq \text{sym2} + du2 * m$

shows $d1 * (du1 * m + 1) + \text{sym1} < d2 * (du2 * m + 1) + \text{sym2}$
proof –
have $1 + (d1 * (du1 * m + 1) + \text{sym1}) \leq 1 + (d1 * (du2 * m + 1) + (\text{sym2} + du2 * m))$
by (*intro add-mono mult-mono, insert assms, auto*)
also have $\dots = (1 + d1) * (du2 * m + 1) + \text{sym2}$

by (*simp add: algebra-simps*)
 also have $\dots \leq d2 * (du2 * m + 1) + sym2$
 by (*intro mult-mono add-mono, insert assms, auto*)
 finally show *?thesis* by *simp*
 qed

lemma *measure-pat-poly-meas-diff*: **assumes** *meas-diff* $p < meas-diff\ p'$
 and *meas-dupl* $p \leq meas-dupl\ p'$
 and *meas-tsymbols* $p \leq meas-tsymbols\ p' + meas-dupl\ p' * m$
shows *measure-pat-poly* $c\ p < measure-pat-poly\ c\ p'$
 unfolding *measure-pat-poly-def*
 by (*rule measure-expr-decrease, insert assms, auto*)

lemma *measure-pat-poly-num-syms*: **assumes** *meas-diff* $p \leq meas-diff\ p'$
 and *meas-dupl* $p \leq meas-dupl\ p'$
 and *meas-tsymbols* $p < meas-tsymbols\ p'$
shows *measure-pat-poly* $c\ p < measure-pat-poly\ c\ p'$
 unfolding *measure-pat-poly-def*
 by (*intro add-le-less-mono mult-mono, insert assms, auto*)

definition *rel-pat* :: $(f, v, s)pat\text{-}problem\text{-}mset\ rel\ (\prec \prec)$ **where**
 $(\prec) = inv\text{-}image\ (\{(x, y). x < y\} < *lex* > \{(x, y). x < y\} < *lex* > \{(x, y). x < y\})$
 $(\lambda mp. (meas\text{-}diff\ mp, meas\text{-}finvars\ mp, meas\text{-}tsymbols\ mp))$

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

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

definition *rel-pat'* :: $(f, v, s)pat\text{-}problem\text{-}mset\ rel$ **where**
 $rel\text{-}pat' = inv\text{-}image\ (\{(x, y). x < y\} < *lex* > \{(x, y). x < y\} < *lex* > \{(x, y). x < y\} < *lex* > \{(x, y). x < y\})$
 $(\lambda mp. (meas\text{-}diff\ mp, meas\text{-}finvars\ mp, meas\text{-}tsymbols\ mp, meas\text{-}setsize\ mp))$

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

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

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

lemma *wf-rel-pat'*: *wf rel-pat'*
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 *rel-pat-sub-rel-pat'*: *rel-pat* \subseteq *rel-pat'*
unfolding *rel-pat-def rel-pat'-def* **by** *auto*

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-tsymbols-def meas-setsizes-def*
meas-finvars-mp-def meas-diff-mp-def meas-dupl-def

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

lemma *rel-mp-sub*: $\{\# \text{ add-mset } p \text{ } mp\#\} \succ \{\# mp \#\}$
proof –
let *?mp' = add-mset p mp*
have *mp* $\subseteq_{\#}$ *?mp'* **by** *auto*
from *meas-finvars-mp-mono[OF tvars-match-mono[OF this]]*
show *?thesis* **unfolding** *meas-def rel-pat-def num-tsyms-mp-def* **by** (*cases p, auto*)
qed

lemma *mp-step-tsyms-mp-psubset*:
fixes *mp* :: (*f, 'v, 's*) *match-problem-mset*
assumes *mp* \rightarrow_m *mp'*
shows *tsyms-mp mp'* $\subset_{\#}$ *tsyms-mp mp*
using *assms*
proof *cases*
case *: (*match-decompose f ts g ls mp'*)
hence *len: length ls = length ts* **by** *auto*
have *tsyms-mp mp' = tsyms-mp mp'' + ($\sum p \in \# \text{ mset } (\text{zip } ts \text{ } ls). \text{syms-term } (fst p)$)*

```

    unfolding tsyms-mp-def * o-def by auto
  also have  $(\sum p \in \# \text{ mset } (\text{zip } ts \text{ } ls). \text{ syms-term } (fst \text{ } p)) = (\sum t \in \# \text{ mset } ts. \text{ syms-term } t)$ 
    using len proof (induct ts arbitrary: ls)
    case (Cons t ts ls)
    thus ?case by (cases ls, auto)
  qed auto
  also have  $\text{tsyms-mp } mp'' + \dots \subset \# \text{ tsyms-mp } mp'' + \text{ syms-term } (Fun \text{ } f \text{ } ts)$ 
    by auto
  also have  $\dots = \text{tsyms-mp } mp$ 
    unfolding tsyms-mp-def * by auto
  finally show ?thesis .
next
  case *: (match-decompose' mp1 y f n mp2 ys)
  let ?Var = Var :: 'v  $\Rightarrow$  ('f, 'v) term
  from * obtain t l mp1' where mp1: mp1 = add-mset (t,l) mp1' by (cases mp1, auto)
  from *(3)[of t l] *(6) mp1 obtain ts where t: t = Fun f ts and lens: length ts = length ys
    by (cases t, auto)
  let ?TS =  $(\sum \# \text{ (image-mset syms-term (mset } ts)))$ 
  have  $\text{tsyms-mp } mp = \text{tsyms-mp } mp1 + \text{tsyms-mp } mp2$  unfolding * tsyms-mp-def by auto
  also have  $\dots = \text{syms-term } t + \text{tsyms-mp } mp1' + \text{tsyms-mp } mp2$  unfolding mp1 tsyms-mp-def o-def by auto
  also have  $\text{syms-term } t = \text{add-mset } (Inr \text{ } f) \text{ } ?TS$  unfolding t by auto
  finally have mp:  $\text{tsyms-mp } mp \supset \# ?TS + \text{tsyms-mp } mp1' + \text{tsyms-mp } mp2$  by auto

  have  $\text{tsyms-mp } mp' = \text{sum-ms } (\text{syms-term } o \text{ } fst) (\text{sum-ms } (\lambda (t, l). \text{ mset } (\text{zip } (args \text{ } t) (\text{map } ?Var \text{ } ys)))) \text{ } mp1$ 
    +  $\text{tsyms-mp } mp2$  unfolding * tsyms-mp-def
    by simp
  also have  $\text{image-mset } (\text{syms-term } o \text{ } fst) (\text{sum-ms } (\lambda (t, l). \text{ mset } (\text{zip } (args \text{ } t) (\text{map } ?Var \text{ } ys)))) \text{ } mp1$ 
    =  $(\text{sum-ms } (\lambda (t, l). \text{ image-mset } (\text{syms-term } o \text{ } fst) (\text{mset } (\text{zip } (args \text{ } t) (\text{map } ?Var \text{ } ys)))) \text{ } mp1)$ 
    by (induct mp1, auto)
  also have  $\dots = \text{sum-ms } (\lambda p. \text{ image-mset } \text{ syms-term } (\text{mset } (\text{args } (fst \text{ } p)))) \text{ } mp1$ 
  proof (rule arg-cong[of - - sum-mset], rule image-mset-cong, goal-cases)
  case (1 p)
  {
    fix t l
    assume (t,l)  $\in \# \text{ } mp1$ 
    from *(3)[OF this] *(6) obtain ts where args: args t = ts and len: length ts = length ys by (cases t, auto)
    have  $\text{image-mset } (\text{syms-term } o \text{ } fst) (\text{mset } (\text{zip } (args \text{ } t) (\text{map } ?Var \text{ } ys))) = \text{image-mset } \text{ syms-term } (\text{mset } (args \text{ } t))$ 
      unfolding args using len

```

```

    proof (induct ts arbitrary: ys)
      case (Cons t ts ys)
      thus ?case by (cases ys, auto)
    qed auto
  }
  thus ?case using 1 by (cases p, auto)
qed
  finally have mp': tsyms-mp mp' = ?TS +  $\sum \# (\sum p \in \# mp1'. \text{image-mset } \text{syms-term } (\text{mset } (\text{args } (\text{fst } p)))) + \text{tsyms-mp } mp2$ 
    by (auto simp: mp1 t)

  have mp1':  $\sum \# (\sum p \in \# mp1'. \text{image-mset } \text{syms-term } (\text{mset } (\text{args } (\text{fst } p)))) \subseteq \# \text{tsyms-mp } mp1'$ 
    unfolding tsyms-mp-def image-mset.compositionality sum-ms-image o-def
  proof (induct mp1')
    case (add tl mp)
    then obtain t l where tl: tl = (t,l) by force
    show ?case unfolding tl
      apply simp
      apply (rule subset-mset.add-mono)
      subgoal by (cases t, auto)
      subgoal using add by auto
    done
  qed auto
  show ?thesis unfolding mp' using mp mp1'
    subset-mset.dual-order.strict-trans2 by fastforce
next
  case *: (match-match x t)
  show ?thesis unfolding * by (rule tsyms-mp-strict-mono, auto)
next
  case *: (match-duplicate pair mp)
  show ?thesis unfolding * by (rule tsyms-mp-strict-mono, auto)
qed

```

lemma *tvars-match-tsyms-mp*: $\text{tvars-match } (\text{mp-mset } mp) = \{ x. \text{Inl } x \in \# \text{tsyms-mp } mp \}$
 unfolding tsyms-mp-def tvars-match-def o-def
 by (force simp: vars-term-syms-term)

lemma *max-dupl-mp-mono*: **assumes** $\text{tsyms-mp } mp \subseteq \# \text{tsyms-mp } mp'$
shows $\text{max-dupl-mp } mp \leq \text{max-dupl-mp } mp'$
 unfolding max-dupl-mp-def
proof (rule Max-le-MaxI, goal-cases)
 case 1
 show ?case by (auto intro: tvars-match-fin)
next
 case 2
 show ?case by auto

```

next
  case 3
  show ?case by (auto intro: tvars-match-fin)
next
  case (4 d)
  show ?case
  proof (cases d = 0)
    case True
    thus ?thesis by auto
  next
    case False
    let ?Inl = Inl :: nat × 's ⇒ nat × 's + 'f
    from 4 False obtain x where x: x ∈ tvars-match (mp-mset mp)
    and d: d = (∑ t∈#image-mset fst mp. count (syms-term t) (?Inl x)) by auto
    have eq: (∑ t∈#image-mset fst mp. count (syms-term t) (?Inl x)) = count
      (tsyms-mp mp) (?Inl x) for
      mp :: (('f, nat × 's) term × ('f, 'v) term) multiset
    unfolding tsyms-mp-def o-def by (induct mp, auto)
    from assms x have x': x ∈ tvars-match (mp-mset mp') unfolding tvars-match-tsyms-mp

    using set-mset-mono by auto
    have d ≤ (∑ t∈#image-mset fst mp'. count (syms-term t) (?Inl x))
    unfolding d eq using assms by (rule mset-subset-eq-count)
    with x' show ?thesis by auto
  qed
qed

lemma mp-step-mset-meas-max-dupl: assumes mp →m mp'
  shows max-dupl-mp mp' ≤ max-dupl-mp mp
  by (rule max-dupl-mp-mono, insert mp-step-tsyms-mp-psubset[OF assms], auto)

lemma mp-step-mset-meas-finvars: assumes mp →m mp'
  shows meas-finvars-mp mp' ≤ meas-finvars-mp mp
proof (rule meas-finvars-mp-mono)
  from mp-step-tsyms-mp-psubset[OF assms]
  have tsyms-mp mp' ⊆# tsyms-mp mp by auto
  thus tvars-match (mp-mset mp') ⊆ tvars-match (mp-mset mp)
  unfolding tvars-match-tsyms-mp
  by (meson Collect-mono set-mset-mono subsetD)
qed

lemma mp-step-mset-num-tsyms-mp: assumes mp →m mp'
  shows num-tsyms-mp mp' < num-tsyms-mp mp
proof -
  from mp-step-tsyms-mp-psubset[OF assms]
  have tsyms-mp mp' ⊂# tsyms-mp mp by auto
  thus ?thesis by (rule tsyms-mp-subset-num-tsyms)
qed

```

lemma *mp-step-mset-meas-diff-mp*:
fixes $mp :: ('f, 'v, 's)$ *match-problem-mset*
assumes $mp \rightarrow_m mp'$
shows $meas\text{-}diff\text{-}mp\ mp' \leq meas\text{-}diff\text{-}mp\ mp$
using *assms*
proof *cases*
case *: (*match-decompose* $f\ ts\ g\ ls\ mp''$)
have id : (*case* *case* x of $(x, y) \Rightarrow (y, x)$ of $(t, l) \Rightarrow f\ t\ l$) = (*case* x of $(a, b) \Rightarrow f$
 $b\ a$) **for**
 $x :: ('f, 'v)$ *Term.term* $\times ('f, nat \times 's)$ *Term.term* **and** $f :: - \Rightarrow - \Rightarrow nat$
by (*cases* x , *auto*)
show $meas\text{-}diff\text{-}mp\ mp' \leq meas\text{-}diff\text{-}mp\ mp$
unfolding *meas-def* * **using** *(3)
by (*auto simp: sum-mset-sum-list[symmetric]* *zip-commute[of ts ls]* *image-mset.compositionality*
o-def id)
next
case *: (*match-decompose'* $mp1\ y\ f\ n\ mp2\ ys$)
let $?Var = Var :: 'v \Rightarrow ('f, 'v)$ *term*
have $meas\text{-}diff\text{-}mp\ mp' \leq meas\text{-}diff\text{-}mp\ mp$
 $\longleftrightarrow (\sum (ti, yi) \in \#(\sum (t, l) \in \#mp1. mset (zip (args\ t) (map\ ?Var\ ys)))) . fun\text{-}diff$
 $yi\ ti$
 $\leq (\sum (t, l) \in \#mp1. fun\text{-}diff\ l\ t)$ (**is** $- \longleftrightarrow ?sum \leq -$)
unfolding * *meas-diff-mp-def* **by** *simp*
also have $?sum = 0$
by (*intro sum-mset.neutral ballI, auto simp: set-zip*)
finally show $?thesis$ **by** *simp*
qed (*auto simp: meas-def*)

lemma *rel-mp-mp-step-mset*:
fixes $mp :: ('f, 'v, 's)$ *match-problem-mset*
assumes $step: mp \rightarrow_m mp'$
shows $\{\#mp\#\} > \{\#mp'\#\}$
proof –
from
 $mp\text{-}step\text{-}mset\text{-}meas\text{-}finvars[OF\ step]$
 $mp\text{-}step\text{-}mset\text{-}num\text{-}tsyms\text{-}mp[OF\ step]$
 $mp\text{-}step\text{-}mset\text{-}meas\text{-}diff\text{-}mp[OF\ step]$
show $?thesis$ **by** (*auto simp: rel-pat-def meas-diff-def meas-finvars-def meas-tsymbols-def*)
qed

lemma *mp-step-measure-pat-poly*:
fixes $mp :: ('f, 'v, 's)$ *match-problem-mset*
assumes $step: mp \rightarrow_m mp'$
shows $measure\text{-}pat\text{-}poly\ c\ (add\text{-}mset\ mp\ p) > measure\text{-}pat\text{-}poly\ c\ (add\text{-}mset\ mp'\ p)$
proof (*rule measure-pat-poly-num-syms*)
show $meas\text{-}diff\ (add\text{-}mset\ mp'\ p) \leq meas\text{-}diff\ (add\text{-}mset\ mp\ p)$
using *mp-step-mset-meas-diff-mp[OF step]* **unfolding** *meas-diff-def* **by** *auto*

show $\text{meas-dupl } (\text{add-mset } mp' p) \leq \text{meas-dupl } (\text{add-mset } mp p)$
using $\text{mp-step-mset-meas-max-dupl}[OF \text{ step}]$ **unfolding** meas-dupl-def **by** *auto*
show $\text{meas-tsymbols } (\text{add-mset } mp' p) < \text{meas-tsymbols } (\text{add-mset } mp p)$
using $\text{mp-step-mset-num-tsymb-mp}[OF \text{ step}]$ **unfolding** meas-tsymbols-def **by**
auto
qed

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

lemma meas-sub : **assumes** sub : $p' \subseteq\# p$
shows $\text{meas-diff } p' \leq \text{meas-diff } p$
 $\text{meas-finvars } p' \leq \text{meas-finvars } p$
 $\text{meas-tsymbols } p' \leq \text{meas-tsymbols } p$
 $\text{meas-dupl } p' \leq \text{meas-dupl } p$

proof –
from sub **obtain** p'' **where** p : $p = p' + p''$ **by** ($\text{metis subset-mset.less-eqE}$)
show $\text{meas-diff } p' \leq \text{meas-diff } p$ $\text{meas-finvars } p' \leq \text{meas-finvars } p$
 $\text{meas-tsymbols } p' \leq \text{meas-tsymbols } p$ $\text{meas-dupl } p' \leq \text{meas-dupl } p$
unfolding $\text{meas-def } p$ **by** *auto*
qed

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

lemma $\text{max-size-term-of-sort}$: **assumes** sS : $s \in S$ **and** inf : $\neg \text{inf-sort } s$
shows $\exists t. t : s \text{ in } \mathcal{T}(C) \wedge \text{max-size } s = \text{size } t \wedge (\forall t'. t' : s \text{ in } \mathcal{T}(C) \longrightarrow \text{size } t' \leq \text{size } t)$
proof –
let $?set = \lambda s. \text{size } \{t. t : s \text{ in } \mathcal{T}(C)\}$
have m : $\text{max-size } s = \text{Maximum } (?set s)$ **unfolding** $\text{o-def max-size-def}$ **using**
 $\text{inf } sS$ **by** *auto*
from $\text{inf inf-sort-not-bdd}[OF sS]$ **have** $\text{bdd-above } (?set s)$ **by** *auto*
moreover **have** $?set s \neq \{\}$ **by** (auto intro! : $\text{sorts-non-empty } sS$)
ultimately **have** $\text{has-Maximum } (?set s)$ **by** ($\text{rule bdd-above-has-Maximum-nat}$)
from $\text{has-MaximumD}[OF \text{ this, folded } m]$ **show** $?thesis$ **by** *auto*

qed

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

lemma *finite-sort-size*: **assumes** *c*: $c : \text{map snd } vs \rightarrow s \text{ in } C$
 and *inf*: $\neg \text{inf-sort } s$
shows $\text{sum } (\text{max-size } o \text{ snd}) (\text{set } vs) < \text{max-size } s$
proof –
 from *c* **have** *vsS*: $\text{insert } s (\text{set } (\text{map snd } vs)) \subseteq S$ **using** *C-sub-S*
 by (*metis (mono-tags)*)
 hence *sS*: $s \in S$ **by auto**
 let *?m* = *max-size s*
 show *?thesis*
 proof (*cases* $\exists v \in \text{set } vs. \text{inf-sort } (\text{snd } v)$)
 case *True*
 {
 fix *v*
 assume $v \in \text{set } vs$
 with *vsS* **have** $v : \text{snd } v \in S$ **by auto**
 note *sorts-non-empty*[*OF this*]
 }
 hence $\forall v. \exists t. v \in \text{set } vs \longrightarrow t : \text{snd } v \text{ in } \mathcal{T}(C)$ **by auto**
 from *choice*[*OF this*] **obtain** *t* **where**
 $t : \bigwedge v. v \in \text{set } vs \implies t v : \text{snd } v \text{ in } \mathcal{T}(C)$ **by blast**
 from *True vsS* **obtain** *vl* **where** $vl : vl \in \text{set } vs$ **and** *vlS*: $\text{snd } vl \in S$ **and** *inf-vl*:
 $\text{inf-sort } (\text{snd } vl)$ **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**
 $tl : \text{snd } vl \text{ in } \mathcal{T}(C)$ **and** *large*: $?m \leq \text{size } tl$ **by fastforce**
 let *?t* = *Fun c (map (λ v. if v = vl then tl else t v) vs)*
 have *?t* : $s \text{ in } \mathcal{T}(C)$
 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 $v : v \in \text{set } vs$
 with *False* **have** *inf*: $\neg \text{inf-sort } (\text{snd } v)$ **by auto**
 from *vsS v* **have** $\text{snd } v \in S$ **by auto**
 from *max-size-term-of-sort*[*OF this inf*]
 have $\exists t. t : \text{snd } v \text{ in } \mathcal{T}(C) \wedge \text{size } t = \text{max-size } (\text{snd } v)$ **by auto**

by *auto*
from *choice[OF this]* **obtain** *t* **where**
 $t : v \in \text{set } vs \implies t v : \text{snd } v \text{ in } \mathcal{T}(C) \wedge \text{size } t = \text{max-size } (\text{snd } v)$
by *blast*
let $?t = \text{Fun } c (\text{map } t \text{ vs})$
have $?t : s \text{ in } \mathcal{T}(C)$
by (*intro Fun-hastypeI[OF c] list-all2-map-map, insert t, auto*)
from *max-size-max[OF sS inf this]*
have $\text{size } ?t \leq \text{max-size } s$.

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

lemma *add-mset-rel-pat*: **assumes** $\text{sub} : mp \neq \{\#\}$
shows $\text{add-mset } mp \succ p$
proof –
from *sub* **obtain** *t l mp'* **where** $mp : mp = \text{add-mset } (t,l) \text{ mp}'$ **by** (*cases mp, auto*)
hence $lt : \text{meas-tsymbols } p < \text{meas-tsymbols } (\text{add-mset } mp \text{ p})$ **unfolding** *meas-def num-tsyms-mp-def* **by** *auto*
from *lt meas-sub[of p add-mset mp p]*
show *?thesis* **unfolding** *rel-pat-def* **by** *auto*
qed

lemma *add-mset-measure-pat-poly*: **assumes** $\text{sub} : mp \neq \{\#\}$
shows $\text{measure-pat-poly } c (\text{add-mset } mp \text{ p}) > \text{measure-pat-poly } c \text{ p}$
proof –
from *sub* **obtain** *t l mp'* **where** $mp : mp = \text{add-mset } (t,l) \text{ mp}'$ **by** (*cases mp, auto*)
hence $lt : \text{meas-tsymbols } p < \text{meas-tsymbols } (\text{add-mset } mp \text{ p})$ **unfolding** *meas-def num-tsyms-mp-def* **by** *auto*
show *?thesis*
by (*rule measure-pat-poly-num-syms[OF - - lt], auto simp: meas-def*)
qed

lemma *meas-dupl-inst*: **fixes** $p :: ('f, 'v, 's) \text{pat-problem-mset}$
assumes $\tau \in \tau s \text{ n } x$
and *disj*: $\text{tvars-disj-pp } \{n..<n + m\} (\text{pat-mset } p)$
shows $\text{meas-dupl } (\text{subst-pat-problem-mset } \tau \text{ p}) \leq \text{meas-dupl } p$

```

proof -
  let ?tau-mset = subst-pat-problem-mset  $\tau :: (f, 'v, 's)$  pat-problem-mset  $\Rightarrow$  -
  let ?tau = subst-match-problem-mset  $\tau :: (f, 'v, 's)$  match-problem-mset  $\Rightarrow$  -
  from assms[unfolded  $\tau$ s-def] obtain f ss
    where f:  $f : ss \rightarrow \text{snd } x \text{ in } C$  and  $\tau$ :  $\tau = \tau c n x (f, ss)$  by auto
  define vs where vs = indexed-from n ss
  define s where s = (Fun f (map Var vs))
  have tau:  $\tau = \text{subst } x s$ 
    using  $\tau$  by (simp add:  $\tau$ c-def s-def vs-def)
  have meas-dupl (?tau-mset p) = sum-ms (max-dupl-mp o ?tau) p
    unfolding meas-dupl-def subst-pat-problem-mset-def o-def image-mset.compositionality
  ..
  also have ...  $\leq$  sum-ms max-dupl-mp p unfolding o-def
  proof (rule sum-mset-mono)
    fix mp
    assume mp:  $mp \in \# p$ 
    show max-dupl-mp (?tau mp)  $\leq$  max-dupl-mp mp
    proof (cases  $x \in \text{tvars-match } (mp\text{-mset } mp)$ )
      case False
      have (?tau mp) = image-mset id mp unfolding subst-match-problem-mset-def
subst-left-def
      proof (rule image-mset-cong, clarsimp, goal-cases)
        case (1 t l)
        with False have  $x \notin \text{vars } t$  unfolding tvars-match-def by force
        thus  $t \cdot \tau = t$  unfolding tau by simp
      qed
    thus ?thesis by simp
  next
  case x: True
  show ?thesis unfolding max-dupl-mp-def
    proof (rule Max-le-MaxI, force intro: tvars-match-fin, force, force intro:
tvars-match-fin, goal-cases)
      case (1 d)
      show ?case
      proof (cases  $d = 0$ )
        case True
        thus ?thesis by blast
      next
      case False
      with 1 obtain y where y:  $y \in \text{tvars-match } (mp\text{-mset } (?tau mp))$ 
        and d:  $d = (\sum t \in \# \text{image-mset } \text{fst } (?tau mp). \text{count } (\text{syms-term } t) (\text{Inl }
y))$ 
        by auto
      have syms-s:  $\text{syms-term } s = \text{add-mset } (\text{Inr } f) (\text{mset } (\text{map } \text{Inl } vs))$  unfolding
s-def
        by (simp, induct vs, auto)
        have tvars-match (mp-mset (?tau mp)) =  $\bigcup (\text{vars } ' \tau ' \text{tvars-match }
(mp\text{-mset } mp))$ 
        unfolding tvars-match-def subst-match-problem-mset-def subst-left-def

```

```

    by (force simp: vars-term-subst)
  also have ... = vars ( $\tau$  x)  $\cup$   $\bigcup$  (vars ' $\tau$ ' (tvars-match (mp-mset mp) -
{x}))
    using x by auto
  also have vars ( $\tau$  x) = set vs unfolding tau s-def by auto
  also have  $\bigcup$  (vars ' $\tau$ ' (tvars-match (mp-mset mp) - {x})) = tvars-match
(mp-mset mp) - {x}
    unfolding tau by (auto simp: subst-def)
  finally have tvars-match (mp-mset (?tau mp)) = set vs  $\cup$  (tvars-match
(mp-mset mp) - {x}) .
    with y have y: y  $\in$  set vs  $\vee$  y  $\in$  tvars-match (mp-mset mp) - {x} by
auto

  define repl :: ('f, nat  $\times$  's) Term.term  $\Rightarrow$  (nat  $\times$  's + 'f) multiset
    where repl t = replicate-mset (count (syms-term t) (Inl x)) (Inl x) for t
  define terms where terms = image-mset fst mp
  let ?add = ( $\sum$  t $\in$ #terms. count (repl t) (Inl y))
  let ?symsy = ( $\sum$  t $\in$ #terms. count (syms-term t) (Inl y))
  let ?symsxy = ( $\sum$  t $\in$ #terms. count (syms-term t) (Inl x) * count (syms-term
s) (Inl y))
  let ?symsx = ( $\sum$  t $\in$ #terms. count (syms-term t) (Inl x))
  have d = ( $\sum$  t $\in$ #terms. count (syms-term (t  $\cdot$   $\tau$ )) (Inl y))
    unfolding d terms-def subst-match-problem-mset-def subst-left-def
    by (induct mp, auto)
  also have ...  $\leq$  ... + ?add by simp
  also have ... =
    ( $\sum$  t $\in$ #terms. count (syms-term (t  $\cdot$   $\tau$ ) + repl t) (Inl y))
    unfolding sum-mset.distrib[symmetric]
    by (rule arg-cong[of - - sum-mset], rule image-mset-cong, simp)
  also have ... = ?symsy + ?symsxy
    unfolding repl-def tau syms-term-subst sum-mset.distrib[symmetric] by
simp

  finally have d: d  $\leq$  ?symsy + ?symsxy .
  show ?thesis
  proof (cases y  $\in$  set vs)
    case False
      hence ?symsxy = 0 unfolding syms-s by auto
      with d have d: d  $\leq$  ?symsy by auto
      from y[simplified] False
      have y  $\in$  tvars-match (mp-mset mp) by auto
      with d show ?thesis unfolding terms-def by auto
    next
      case True
      have dist: distinct vs
        by (simp add: vs-def)
      from split-list[OF True] obtain vs1 vs2 where vs: vs = vs1 @ y # vs2
      by auto
      with dist have nmem: y  $\notin$  set vs1 y  $\notin$  set vs2 by auto
      have count (syms-term s) (Inl y) = 1 unfolding syms-s vs using nmem
        by (auto simp: count-eq-zero-iff)

```

```

with  $d$  have  $d: d \leq ?symsy + ?symsx$  by auto
from True have  $ynm: fst\ y \in \{n..<n + m\}$  using  $m$  [OF f]
  by (simp add: set-zip in-set-indexed-from-eq vs-def)
from  $mp$  have  $mp\text{-mset}\ mp \in pat\text{-mset}\ p$  by auto
from  $assms(2)$  [unfolded tvars-disj-pp-def, rule-format, OF this]  $ynm$ 
have  $y \notin tvars\text{-match}\ (mp\text{-mset}\ mp)$  unfolding tvars-match-def by force
  hence  $y \notin (\bigcup (vars\ 'set\text{-mset}\ terms))$  unfolding tvars-match-def
terms-def
  by auto
  hence  $?symsy = 0$ 
  by (auto simp: count-eq-zero-iff vars-term-syms-term)
  with  $d$  have  $d \leq ?symsx$  by auto
  with  $x$  show  $?thesis$  unfolding terms-def by auto
qed
qed
qed
qed
qed
finally show  $meas\text{-dupl}\ p \geq meas\text{-dupl}\ (?tau\text{-mset}\ p)$  unfolding meas-dupl-def
by auto
qed

```

```

lemma meas-tsymbols-inst: fixes  $p :: ('f, 'v, 's)\ pat\text{-problem}\text{-mset}$ 
  assumes  $\tau \in \tau s\ n\ x$ 
  shows  $meas\text{-tsymbols}\ (subst\text{-pat}\text{-problem}\text{-mset}\ \tau\ p) \leq meas\text{-tsymbols}\ p + meas\text{-dupl}\ p * m$ 
proof -
  let  $?tau\text{-mset} = subst\text{-pat}\text{-problem}\text{-mset}\ \tau :: ('f, 'v, 's)\ pat\text{-problem}\text{-mset} \Rightarrow -$ 
  let  $?tau = subst\text{-match}\text{-problem}\text{-mset}\ \tau :: ('f, 'v, 's)\ match\text{-problem}\text{-mset} \Rightarrow -$ 
  have  $meas\text{-tsymbols}\ (?tau\text{-mset}\ p) = sum\text{-ms}\ (num\text{-tsyms}\text{-mp}\ o\ ?tau)\ p$ 
  unfolding meas-tsymbols-def subst-pat-problem-mset-def o-def image-mset.compositionality
  ..
  also have  $\dots \leq sum\text{-ms}\ (\lambda\ mp.\ num\text{-tsyms}\text{-mp}\ mp + max\text{-dupl}\text{-mp}\ mp * m)\ p$ 
unfolding o-def
  proof (rule sum-mset-mono)
    fix  $mp$ 
    have  $num\text{-tsyms}\text{-mp}\ (?tau\ mp) = sum\text{-ms}\ (num\text{-syms}\ o\ (\lambda\ t.\ t \cdot \tau))\ (image\text{-mset}\ fst\ mp)$ 
    unfolding num-tsyms-mp-def o-def subst-match-problem-mset-def subst-left-def
    by (induct mp, auto)
    also have  $\dots \leq sum\text{-ms}\ (\lambda\ p.\ num\text{-syms}\ p + count\ (syms\text{-term}\ p)\ (Inl\ x) * m)$ 
    (image-mset fst mp)
    unfolding o-def
    by (rule sum-mset-mono[OF num-syms- $\tau$ s[OF assms]])
    also have  $\dots = num\text{-tsyms}\text{-mp}\ mp + sum\text{-ms}\ (\lambda\ p.\ count\ (syms\text{-term}\ p)\ (Inl\ x))\ (image\text{-mset}\ fst\ mp) * m$ 
    unfolding num-tsyms-mp-def o-def
    by (induct mp, auto simp: algebra-simps)
    also have  $\dots \leq num\text{-tsyms}\text{-mp}\ mp + max\text{-dupl}\text{-mp}\ mp * m$ 

```

```

proof (intro add-left-mono mult-right-mono)
show ( $\sum p \in \# \text{image-mset fst mp. count (syms-term } p) (\text{Inl } x) \leq \text{max-dupl-mp}$ 
mp
proof (cases  $x \in \text{tvars-match (mp-mset mp)}$ )
  case True
show ?thesis unfolding max-dupl-mp-def
  by (rule Max-ge, insert True, auto intro: tvars-match-fin)
next
  case False
have ( $\sum p \in \# \text{image-mset fst mp. count (syms-term } p) (\text{Inl } x) = 0$ )
proof (clarsimp)
  fix t l
  assume (t,l)  $\in \#$  mp
  with False have  $x \notin \text{vars } t$  unfolding tvars-match-def by auto
  thus  $\text{count (syms-term } t) (\text{Inl } x) = 0$ 
  unfolding vars-term-syms-term
  by (simp add: count-eq-zero-iff)
qed
thus ?thesis by linarith
qed
qed auto
finally show  $\text{num-tsyzms-mp } (?tau \text{ mp}) \leq \text{num-tsyzms-mp } mp + \text{max-dupl-mp}$ 
mp * m .
qed
also have  $\dots = \text{meas-tsyzms } p + \text{meas-dupl } p * m$  unfolding meas-tsyzms-def
meas-dupl-def
by (induct p, auto simp: algebra-simps)
finally show  $\text{meas-tsyzms } p + \text{meas-dupl } p * m \geq \text{meas-tsyzms } (?tau\text{-mset}$ 
p) .
qed

lemma pp-step-le-size: assumes  $p \Rightarrow_m ps$  and  $p' \in \# ps$ 
shows  $\text{size } p' \leq \text{size } p$ 
using assms by induct (auto simp: subst-pat-problem-mset-def)

lemma decrease-pp-step-mset:
fixes  $p :: ('f, 'v, 's) \text{pat-problem-mset}$ 
assumes  $p \Rightarrow_m ps$ 
and  $p' \in \# ps$ 
shows  $p \succ p' \text{ improved} \implies \text{measure-pat-poly } c \text{ } p > \text{measure-pat-poly } c \text{ } p'$ 
using assms
proof (atomize(full), induct)
  case *: (pat-simp-mp mp mp' p)
  hence  $p': p' = \text{add-mset } mp' \text{ } p$  by auto
  from rel-mp-mp-step-mset[OF *(1)] mp-step-measure-pat-poly[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

```

```

from *(1) have mp: mp ≠ {#} by (cases, auto)
show ?case unfolding p'
  by (intro conjI impI, rule add-mset-rel-pat, rule mp, rule add-mset-measure-pat-poly,
rule mp)
next
case *: (pat-instantiate n mp p x l s y t)
let ?c = c
from *(2) have ∃ s t. (s,t) ∈# mp ∧ (s = Var x ∧ is-Fun t
  ∨ (¬ improved ∧ x ∈ vars s ∧ ¬ inf-sort (snd x)))
proof
  assume *: ¬ improved ∧ (s, Var y) ∈# mp ∧ (t, Var y) ∈# mp ∧ Conflict-Var
s t x ∧ ¬ inf-sort (snd x)
  hence Conflict-Var s t x and ¬ inf-sort (snd x) by auto
  from conflicts(4)[OF this(1)] this(2) *
  show ?thesis by auto
qed auto
then obtain s t where st: (s,t) ∈# mp and choice: s = Var x ∧ is-Fun t ∨ ¬
improved ∧ x ∈ vars s ∧ ¬ inf-sort (snd x)
  by auto
let ?p = add-mset mp p
let ?s = snd x
from *(3) τs-list
obtain τ where τs: τ ∈ τs 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 ⇒ -
let ?tau = subst-match-problem-mset τ :: ('f,'v,'s) match-problem-mset ⇒ -
from τs[unfolded τs-def τc-def]
obtain c sorts where c: c : sorts → ?s in C
  and tau: τ = subst x (Fun c (map Var (indexed-from n sorts)))
  by (clarsimp simp add: τs-def τc-def)
with C-sub-S have sS: ?s ∈ S and sorts: set sorts ⊆ S by auto
define vs where vs = indexed-from n sorts
have τ: τ = subst x (Fun c (map Var vs))
  unfolding tau vs-def by auto
have snd ' vars (τ y) ⊆ insert (snd y) S for y
  using sorts unfolding tau by (auto simp: subst-def set-zip set-conv-nth nth-indexed-from-eq)
hence vars-sort: (a,b) ∈ vars (τ y) ⇒ b ∈ insert (snd y) S for a b y by fastforce

from st obtain mp' where mp: mp = add-mset (s,t) mp' by (rule mset-add)
from choice have ?p > ?tau-mset ?p ∧ (improved → measure-pat-poly ?c ?p
> measure-pat-poly ?c (?tau-mset ?p))
proof
  assume s = Var x ∧ is-Fun t
  then obtain f ts where s: s = Var x and t: t = Fun f ts by (cases t, auto)
  have meas-diff (?tau-mset ?p) =
    meas-diff (?tau-mset (add-mset mp' p)) + fun-diff t (s · τ)
  unfolding meas-def subst-defs subst-left-def mp by simp
  also have ... ≤ meas-diff (add-mset mp' p) + fun-diff t (τ x) using meas-diff-subst-le[of

```

```

τ] s by auto
also have ... < meas-diff (add-mset mp' p) + fun-diff t s
proof (rule add-strict-left-mono)
  have fun-diff t (τ x) < num-funs t
  unfolding tau subst-simps fun-diff.simps
  by (rule fun-diff-num-funs-lt[OF refl], auto simp: t)
  thus fun-diff t (τ x) < fun-diff t s by (auto simp: s t)
qed
also have ... = meas-diff ?p unfolding mp meas-def by auto
finally have md: meas-diff (?tau-mset ?p) < meas-diff ?p .
show ?thesis
proof (intro conjI impI)
  show ?p > ?tau-mset ?p using md unfolding rel-pat-def by auto
  show measure-pat-poly ?c ?p > measure-pat-poly ?c (?tau-mset ?p)
  proof (rule measure-pat-poly-meas-diff[OF md])
    show meas-dupl ?p ≥ meas-dupl (?tau-mset ?p)
    by (rule meas-dupl-inst[OF τs], insert *, auto)
    show meas-tsymbols ?p + meas-dupl ?p * m ≥ meas-tsymbols (?tau-mset
?p)
    by (rule meas-tsymbols-inst[OF τs])
  qed
qed
next
assume ¬ improved ∧ x ∈ vars s ∧ ¬ inf-sort (snd x)
hence x: x ∈ vars s and inf: ¬ inf-sort (snd x) and impr: ¬ improved by auto
from meas-diff-subst-le[of τ]
have fd: meas-diff p' ≤ meas-diff ?p unfolding p' .
  have meas-finvars (?tau-mset ?p) = meas-finvars (?tau-mset {#mp#}) +
meas-finvars (?tau-mset p)
  unfolding subst-defs meas-def by auto
  also have ... < meas-finvars {#mp#} + meas-finvars p
  proof (rule add-less-le-mono)
    have vars-τ-var: vars (τ y) = (if x = y then set vs else {y}) for y unfolding
τ subst-def by auto
    have vars-τ: vars (t · τ) = vars t - {x} ∪ (if x ∈ vars t then set vs else {})
for t
    unfolding vars-term-subst image-comp o-def vars-τ-var by auto
    have tvars-match-subst: tvars-match (mp-mset (?tau mp)) =
tvars-match (mp-mset mp) - {x} ∪ (if x ∈ tvars-match (mp-mset mp)
then set vs else {}) for mp
    unfolding subst-defs subst-left-def tvars-match-def
    by (auto simp: vars-τ split: if-splits prod.split)
    have id1: meas-finvars (?tau-mset {#mp#}) = (∑ x ∈ tvars-match (mp-mset
(?tau mp)). max-size (snd x)) for mp
    unfolding meas-def subst-defs by auto
    have id2: meas-finvars {#mp#} = (∑ x ∈ tvars-match (mp-mset mp). max-size
(snd x))
    for mp :: ('f, 'v, 's) match-problem-mset
    unfolding meas-def subst-defs by simp

```

```

have eq:  $x \notin \text{tvars-match } (mp\text{-mset } mp) \implies \text{meas-finvars } (?tau\text{-mset } \{\# mp \# \}) = \text{meas-finvars } \{\# mp \#\}$  for mp
unfolding id1 id2 by (rule sum.cong[OF - refl], auto simp: tvars-match-subst)
{
  fix mp :: ('f,'v,'s) match-problem-mset

  assume xmp:  $x \in \text{tvars-match } (mp\text{-mset } mp)$ 
  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  $\cup$  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  $\circ$  snd) (set vs  $\cup$  Mp) (is - = sum ?size -)
    unfolding id1 id2 using 1 by auto
  also have ...  $\leq$  sum ?size (set vs) + sum ?size Mp by (rule sum-union-le-nat)
  also have ...  $<$  ?size x + sum ?size Mp
  proof -
    have sS: ?s  $\in S$  by fact
    have sorts: sorts = map snd vs unfolding vs-def by (intro nth-equalityI,
auto)
    have sum ?size (set vs)  $<$  ?size x
      using finite-sort-size[OF c[unfolded sorts] inf] by auto
    thus ?thesis by auto
  qed
  also have ... = meas-finvars {#mp#} unfolding id2 2 using fin xMp by
auto
  finally have meas-finvars (?tau-mset {# mp #})  $<$  meas-finvars {#mp#}
.
} note less = this
have le: meas-finvars (?tau-mset {# mp #})  $\leq$  meas-finvars {#mp#} for
mp
  using eq[of mp] less[of mp] by linarith

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

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

```

```

qed
thus ?case unfolding  $p'$  .
next
  case *: (pat-remove-pp  $p$ )
  thus ?case by auto
next
  case *: (pat-inf-var-conflict  $pp$   $pp'$ )
  hence  $p'$ :  $p' = pp'$  by auto
  have  $p' \subseteq_{\#} pp + pp'$  unfolding  $p'$  by auto
  note mono = meas-sub[OF this]
  from *(2) obtain  $mp$   $pp2$  where  $pp$ :  $pp = add-mset\ mp\ pp2$  by (cases  $pp$ , auto)
  with *(1) have inf-var-conflict (mp-mset  $mp$ ) by auto
  hence  $mp \neq \{\#\}$  unfolding inf-var-conflict-def by auto
  hence meas-tsymbols  $p' < meas-tsymbols\ (pp + pp')$  unfolding  $p'$   $pp$  using
num-syms-pos
  by (cases  $mp$ , auto simp: meas-tsymbols-def num-tsyms-mp-def)
  thus ?case using mono by (auto simp: rel-pat-def intro: measure-pat-poly-num-syms)
qed

```

finally: the transformation is terminating w.r.t. (\succ_{mul})

```

lemma rel-P-trans:
  assumes  $P \Rightarrow_m P'$ 
  shows  $P \succ_{mul} P'$ 
  using assms
proof induct
  case *: (P-failure  $P$ )
  from * have  $P \neq \{\#\}$  by auto
  then obtain  $p' P'$  where  $P$ :  $P = add-mset\ p'\ P'$  by (cases  $P$ , auto)
  show ?case unfolding  $P$  unfolding rel-pats-def
  by (simp add: subset-implies-mult)
next
  case *: (P-simp-pp  $p$   $ps$   $P$ )
  from set-mp[OF rel-pat-sub-rel-pat'] decrease-pp-step-mset[OF *]
  show ?case unfolding rel-pats-def by (metis add-many-mult)
qed

```

termination of the multiset based implementation, poly-complexity in improved case

```

lemma nd-step-le-size: assumes  $(p,q) \in \Rightarrow_{nd}$ 
  shows  $size\ q \leq size\ p$ 
  using assms by induct (auto simp: pp-step-le-size)

```

```

lemma nd-steps-le-size: assumes  $(p,q) \in \Rightarrow_{nd}^*$ 
  shows  $size\ q \leq size\ p$ 
  using assms by (induct, auto dest: nd-step-le-size)

```

```

lemma nd-step-decrease: assumes  $(p,q) \in \Rightarrow_{nd}$ 
  shows  $p \succ q$ 
  improved  $\implies measure-pat-poly\ c\ p > measure-pat-poly\ c\ q$ 

```

proof –
from *assms*
obtain P **where** $p \Rightarrow_m P$ **and** $q \in \# P$
by *cases auto*
from *decrease-pp-step-mset[OF this]*
show $p \succ q$ *improved* \implies *measure-pat-poly c p > measure-pat-poly c q* **by** *auto*
qed

lemma *nd-steps-bound*: **assumes** *improved*
and $(p, q) \in \Rightarrow_{nd} \widetilde{n}$
shows $n \leq$ *measure-pat-poly c p* (**is** ?A)
measure-pat-poly c q + n \leq *measure-pat-poly c p* (**is** ?B)
proof –
from *steps-bound[of - measure-pat-poly c, OF nd-step-decrease(2)][OF - assms(1)]*
assms(2)]
have *measure-pat-poly c q + n* \leq *measure-pat-poly c p* **by** *auto*
thus ?A ?B **by** *auto*
qed

theorem *SN-nd-pstep*: $SN \Rightarrow_{nd}$
proof (*rule SN-subset[of {(p,q). p > q}], rule wf-imp-SN*)
show *wf* ($\{(p, q). p > q\}^{-1}$) **using** *wf-rel-pat*
by (*simp add: converse-unfold*)
qed (*insert nd-step-decrease, auto*)

theorem *SN-P-step*: $SN \Rightarrow$
proof –
have *sub*: $\Rightarrow \subseteq \prec_{mul} \widehat{-1}$
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

6.4 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*: $mp1 \rightarrow_s mp2 \implies mp1 = mp1' \implies mp2 = mp2' \implies mp1' \rightarrow_s mp2'$ **by** *auto*

lemma *mp-step-mset-mp-trans*: $mp \rightarrow_m mp' \implies mp\text{-mset } mp \rightarrow_s mp\text{-mset } mp'$
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*: $mp\text{-fail } mp \implies mp = mp' \implies mp\text{-fail } mp' \text{ by } auto$

lemma *match-fail-mp-fail*: $match\text{-fail } mp \implies mp\text{-fail } (mp\text{-mset } mp)$

```

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 *pp-step-set-cong*: $P \Rightarrow_s Q \implies P = P' \implies Q = Q' \implies P' \Rightarrow_s Q' \text{ by } auto$

lemma *p-step-mset-imp-set*: **assumes** $p \Rightarrow_m Q$

shows $pat\text{-mset } p \Rightarrow_s pats\text{-mset } Q$

using *assms*

proof –

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]

show ?thesis **using** *assms(1) unfolding conv*

proof *induction*

case (*pat-remove-pp p*)

show ?case **unfolding** *conv using pp-success by auto*

next

case *: (*pat-simp-mp mp mp' p*)

from *pp-simp-mp[OF mp-step-mset-mp-trans[OF *]]*

show ?case **by** *auto*

next

case *: (*pat-remove-mp mp p*)

from *pp-remove-mp[OF match-fail-mp-fail[OF *]]*

show ?case **by** *simp*

next

case *: (*pat-instantiate n mp p x l s y t*)

```

from *(2) have  $x \in \text{tvars-match } (mp\text{-mset } mp)$ 
  using  $\text{conflicts}(4)[\text{of } s \ t \ x]$  unfolding  $\text{tvars-match-def}$ 
  by  $(\text{auto intro!}:\text{term.set-intros}(3))$ 
hence  $x: x \in \text{tvars-pat } (\text{pat-mset } (\text{add-mset } mp \ p))$  unfolding  $\text{tvars-pat-def}$ 
  using *(2) by  $\text{auto}$ 
show  $?case$  unfolding  $\text{conv } \tau s\text{-list}$ 
  apply  $(\text{rule } pp\text{-step-set-cong}[OF \ pp\text{-instantiate}[OF \ *(1) \ x]])$ 
  by  $(\text{unfold } \text{conv } \text{subst-defs } \text{set-map } \text{image-comp}, \text{ auto})$ 
next
  case *:  $(\text{pat-inf-var-conflict } pp \ pp')$ 
  from  $pp\text{-inf-var-conflict}[OF \ *(1), \text{ of } \text{pat-mset } pp']$ 
  have  $\text{pat-mset } (pp + pp') \Rightarrow_s \{\text{pat-mset } pp'\}$ 
  using * by  $(\text{auto simp: } \text{tvars-pat-def } \text{image-Un})$ 
  thus  $?case$  by  $\text{auto}$ 
qed
qed

lemma  $pp\text{-step-mset-pcorrect}: p \Rightarrow_m P' \Longrightarrow \text{wf-pat } (\text{pat-mset } p) \Longrightarrow$ 
 $\text{pat-complete } C \ (\text{pat-mset } p) = \text{pats-complete } C \ (\text{pats-mset } P')$ 
by  $(\text{rule } pp\text{-step-pcorrect}[OF \ p\text{-step-mset-imp-set}])$ 

lemma  $P\text{-step-mset-imp-set}: \text{assumes } P \Rightarrow_m Q$ 
  shows  $\text{pats-mset } P \Rightarrow_s \text{pats-mset } Q$ 
  using  $\text{assms}$ 
proof  $(\text{induction})$ 
  case *:  $(P\text{-failure } P)$ 
  from *(1) show  $?case$ 
  by  $(\text{induct}, \text{ auto intro: } P\text{-fail})$ 
next
  case  $(P\text{-simp-pp } pp \ pp' \ P)$ 
  from  $P\text{-simp}[OF \ p\text{-step-mset-imp-set}[OF \ \text{this}]]$ 
  show  $?case$  by  $(\text{simp add: } \text{image-Un})$ 
qed

lemma  $nd\text{-step-mset-pcorrect}: \text{assumes } p \notin NF \Rightarrow_{nd} \text{wf-pat } (\text{pat-mset } p)$ 
  shows  $\text{pat-complete } C \ (\text{pat-mset } p) \longleftrightarrow (\forall q. (p,q) \in \Rightarrow_{nd} \longrightarrow \text{pat-complete } C \ (\text{pat-mset } q))$ 
proof  $-$ 
  from  $\text{assms}(1)$  obtain  $q$  where  $(p,q) \in \Rightarrow_{nd}$  by  $\text{auto}$ 
  then obtain  $Q$  where  $p \Rightarrow_m Q$  by  $\text{cases}$ 
  from  $pp\text{-step-mset-pcorrect}[OF \ \text{this } \text{assms}(2)]$   $pp\text{-nd-step-mset.intros}[OF \ \text{this}]$ 
  have  $(\forall q. (p,q) \in \Rightarrow_{nd} \longrightarrow \text{pat-complete } C \ (\text{pat-mset } q)) \Longrightarrow \text{pat-complete } C \ (\text{pat-mset } p)$  by  $\text{simp}$ 
  moreover {
    fix  $q$ 
    assume  $(p,q) \in \Rightarrow_{nd}$ 
    then obtain  $Q$  where  $p \Rightarrow_m Q$  and  $q \in \# Q$  by  $\text{cases}$ 
    from  $pp\text{-step-mset-pcorrect}[OF \ \text{this}(1) \ \text{assms}(2)]$   $pp\text{-nd-step-mset.intros}[OF \ \text{this}]$   $\text{this}(2)$ 
  }

```

have $pat\text{-complete } C (pat\text{-mset } p) \implies pat\text{-complete } C (pat\text{-mset } q)$ **by** *auto*
}
ultimately show *?thesis* **by** *blast*
qed

lemma *P-step-pp-trans*: **assumes** $(P, Q) \in \Rightarrow$
shows $pat\text{-mset } P \Rightarrow_s pat\text{-mset } Q$
by (*rule P-step-mset-imp-set, insert assms, unfold P-step-def, auto*)

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

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

lemma *nd-step-to-P-step*: **assumes** $(p, q) \in \Rightarrow_{nd}$
shows $\exists Q. add\text{-mset } p P \Rightarrow_m add\text{-mset } q Q$
using *assms*
proof *cases*
case $(1 Q)$
then show *?thesis* **using** *P-simp-pp*[*of p Q P*]
by (*metis mset-add union-iff*)
qed

lemma *nd-steps-to-P-steps*: **assumes** $(p, q) \in \Rightarrow_{nd}^*$
shows $\exists Q. (\Rightarrow_m)^{**} (add\text{-mset } p P) (add\text{-mset } q Q)$
using *assms*
proof (*induct arbitrary: P*)
case $*$: (*step y z*)
from *nd-step-to-P-step*[*OF *(2)*] $*(3)$ **show** *?case*
by (*meson r-into-rtranclp rtranclp-trans*)
qed *auto*

lemma *P-step-to-nd-step*: **assumes** $P \Rightarrow_m Q$
and $q \in \# Q$ **shows** $\exists p \in \# P. (p, q) \in \Rightarrow_{nd}^=$
using *assms(1)*
proof *cases*
case $*$: (*P-simp-pp pp P' P*)

```

with assms have  $q \in\# P' \vee q \in\# P$  by auto
thus ?thesis
proof
  assume  $q \in\# P$ 
  thus ?thesis using * by auto
next
  assume  $q \in\# P'$ 
  with * have  $(pp, q) \in \Rightarrow_{nd}$  by (intro pp-nd-step-mset.intros)
  with * show ?thesis by auto
qed
qed (insert assms, auto)

```

```

lemma P-steps-to-nd-steps: assumes  $(\Rightarrow_m)^{**} P Q$ 
and  $q \in\# Q$  shows  $\exists p \in\# P. (p, q) \in \Rightarrow_{nd}^*$ 
using assms
proof (induct arbitrary: q)
  case *: (step Q R r)
  from P-step-to-nd-step[OF *(2,4)]
  obtain  $q$  where  $q \in\# Q$  and  $(q, r) \in \Rightarrow_{nd}^=$  by auto
  from *(3)[OF this(1)] this(2) show ?case
  by (metis (no-types, lifting) UnE pair-in-Id-conv rtrancl.rtrancl-into-rtrancl)
qed auto

```

```

lemma nd-steps-fail-iff-Psteps-fail:  $(p, \{\#\}) \in \Rightarrow_{nd}^* \longleftrightarrow (\Rightarrow_m)^{**} \{\#p\}$  bottom-mset
proof
  assume  $(p, \{\#\}) \in \Rightarrow_{nd}^*$ 
  from nd-steps-to-P-steps[OF this] obtain  $P$ 
  where steps:  $(\Rightarrow_m)^{**} \{\#p\}$  (add-mset  $\{\#\}$   $P$ ) by auto
  from P-failure[of P] have  $(\Rightarrow_m)^{**}$  (add-mset  $\{\#\}$   $P$ ) bottom-mset
  by (cases add-mset  $\{\#\}$   $P =$  bottom-mset, auto)
  with steps show  $(\Rightarrow_m)^{**} \{\#p\}$  bottom-mset by simp
next
  assume  $(\Rightarrow_m)^{**} \{\#p\}$  bottom-mset
  from P-steps-to-nd-steps[OF this, of  $\{\#\}$ ]
  show  $(p, \{\#\}) \in \Rightarrow_{nd}^*$  by auto
qed

```

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

```

theorem P-step:
assumes non-improved:  $\neg$  improved
and wf: wf-pats (pats-mset  $P$ ) and NF:  $(P, Q) \in \Rightarrow^!$ 
shows  $Q = \{\#\} \wedge$  pats-complete  $C$  (pats-mset  $P$ ) — either the result is and input P is complete
   $\vee Q =$  bottom-mset  $\wedge \neg$  pats-complete  $C$  (pats-mset  $P$ ) — or the result = bot and P is not complete
proof —
  from NF have steps:  $(P, Q) \in \Rightarrow^*$  and NF:  $Q \in$  NF P-step by auto

```

from P -steps-pcorrect[OF wf steps]
have wf : wf-pats (pats-mset Q) **and**
 $sound$: pats-complete C (pats-mset P) = pats-complete C (pats-mset Q)
by blast+
from P -step-NF[OF non-improved wf NF] **have** $Q \in \{\#\}, bottom\text{-mset}\}$.
thus ?thesis **unfolding** $sound$ **by** auto
qed

theorem nd-pstep:

assumes non-improved: $\neg improved$
and wf : wf-pat (pat-mset p)
shows $\neg pat\text{-complete } C$ (pat-mset p) $\longleftrightarrow (p, \{\#\}) \in \Rightarrow_{nd}^*$
proof –
from wf **have** wf : wf-pats (pats-mset $\{\#p\#$) **by** (auto simp: wf-pats-def)
have $\neg pat\text{-complete } C$ (pat-mset p) $\longleftrightarrow (\{\#p\#$, bottom-mset) $\in \Rightarrow^*$
proof –
{
assume $(\{\#p\#$, $\{\#\{\#\}\#$) $\in \Rightarrow^*$
from P -steps-pcorrect[OF wf this]
have $\neg pat\text{-complete } C$ (pat-mset p)
by auto
} **note** bot = this
from SN- P -step **obtain** Q **where** NF: $(\{\#p\#$, Q) $\in \Rightarrow^!$
by (metis SN-def SN-on-imp-normalizability)
from P -step[OF non-improved wf NF]
have res : $Q = \{\#\} \wedge pat\text{-complete } C$ (pat-mset p) $\vee Q = \{\#\{\#\}\# \wedge \neg$
 $pat\text{-complete } C$ (pat-mset p) **by** auto
hence pcQ : $pat\text{-complete } C$ (pat-mset p) = $(Q = \{\#\})$ **by** auto
from NF **have** $(\{\#p\#$, Q) $\in \Rightarrow^*$ **by** auto
thus ?thesis **unfolding** pcQ **using** res bot **by** auto
qed
also $have \dots \longleftrightarrow (\Rightarrow_m)^{**} (\{\#p\#$) bottom-mset
unfolding P -step-def **by** (meson Enum.rtranclp-rtrancl-eq)
also $have \dots \longleftrightarrow (p, \{\#\}) \in \Rightarrow_{nd}^*$
unfolding nd-steps-fail-iff-Psteps-fail ..
finally **show** ?thesis **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: $(P, Q) \in \Rightarrow^!$
shows pats-complete C (pats-mset P) \longleftrightarrow pats-complete C (pats-mset Q) —
equivalence
 $p \in \# Q \implies finite\text{-constr-form-pat } C$ (pat-mset p) — all remaining problems
are in finite-constr-form
proof –
from NF **have** $steps$: $(P, Q) \in \Rightarrow^*$ **and** NF: $Q \in NF$ 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  $p \in \# Q \implies \text{finite-constr-form-pat } C \text{ (pat-mset } p) .$ 
qed

```

theorem *nd-step-improved*:

```

fixes  $p :: ('f, 'v, 's) \text{ pat-problem-mset}$ 
assumes improved

```

```

and inf: infinite (UNIV :: 'v set)

```

```

and wf: wf-pat (pat-mset p)

```

```

shows  $(p, q) \in \Rightarrow_{nd} \widetilde{\widetilde{n}} \implies n \leq \text{num-syms-pat } p * (\text{num-syms-pat } p * m + 2)$ 
 $(p, q) \in \Rightarrow_{nd} \widetilde{\widetilde{n}} \implies \text{measure-pat-poly } c \ q + n \leq (c + \text{num-syms-pat } p) * (\text{num-syms-pat } p * m + 2)$ 

```

```

 $(p, q) \in \Rightarrow_{nd}^! \implies \text{finite-constr-form-pat } C \text{ (pat-mset } q)$ 

```

```

 $(p, q) \in \Rightarrow_{nd}^* \implies \text{pat-complete } C \text{ (pat-mset } p) \implies \text{pat-complete } C \text{ (pat-mset } q)$ 

```

```

 $\neg \text{pat-complete } C \text{ (pat-mset } p) \implies \exists q. (p, q) \in \Rightarrow_{nd}^! \wedge \neg \text{pat-complete } C \text{ (pat-mset } q)$ 

```

```

 $\neg \text{pat-complete } C \text{ (pat-mset } p) \iff (\exists q. (p, q) \in \Rightarrow_{nd}^! \wedge \neg \text{pat-complete } C \text{ (pat-mset } q))$ 

```

proof –

```

{

```

```

fix  $n \ c$ 

```

```

assume  $(p, q) \in (\Rightarrow_{nd}) \widetilde{\widetilde{n}}$ 

```

```

from nd-steps-bound[OF <improved> this, of c] measure-pat-poly[of c p]

```

```

have res: measure-pat-poly  $c \ q + n \leq (c + \text{num-syms-pat } p) * (\text{num-syms-pat } p * m + 2)$  by auto

```

```

} note measure = this

```

```

from this[of n c] this[of n 0]

```

```

show  $(p, q) \in \Rightarrow_{nd} \widetilde{\widetilde{n}} \implies \text{measure-pat-poly } c \ q + n \leq (c + \text{num-syms-pat } p) * (\text{num-syms-pat } p * m + 2)$ 

```

```

 $(p, q) \in \Rightarrow_{nd} \widetilde{\widetilde{n}} \implies n \leq \text{num-syms-pat } p * (\text{num-syms-pat } p * m + 2)$  by auto

```

```

from wf have wf': wf-pats (pats-mset  $\{\#p\#\})$  by (auto simp: wf-pats-def)

```

```

{

```

```

fix  $q$ 

```

```

assume  $(p, q) \in \Rightarrow_{nd}^*$ 

```

```

from nd-steps-to-P-steps[OF this, of \{\#\}] obtain Q where  $(\Rightarrow_m)^{**} \{\#p\#\}$ 
(add-mset  $q \ Q$ ) by auto

```

```

hence  $(\{\#p\#\}, \text{add-mset } q \ Q) \in \Rightarrow^*$  unfolding P-step-def by (meson Enum.rtranclp-rtrancl-eq)

```

```

from P-steps-pcorrect[OF wf' this] have wfq: wf-pats (pats-mset (add-mset  $q \ Q$ ))

```

```

and pats-complete C (pats-mset  $\{\#p\#\})$  = pats-complete C (pats-mset (add-mset  $q \ Q$ )) by auto

```

```

hence pat-complete C (pat-mset  $p$ )  $\implies$  pat-complete C (pat-mset  $q$ )

```

```

    by (auto simp: wf-pats-def)
  moreover {
    assume  $q \in NF \Rightarrow_{nd}$ 
    from nd-step-NF-fvf[OF <improved> inf - this] wfq
    have finite-constr-form-pat  $C$  (pat-mset  $q$ ) by (auto simp: wf-pats-def)
  }
  ultimately have pat-complete  $C$  (pat-mset  $p$ )  $\implies$  pat-complete  $C$  (pat-mset  $q$ )
     $q \in NF \Rightarrow_{nd} \implies$  finite-constr-form-pat  $C$  (pat-mset  $q$ ) by auto
} note result1 = this
thus result1b:  $(p, q) \in \Rightarrow_{nd}^! \implies$  finite-constr-form-pat  $C$  (pat-mset  $q$ ) for  $q$  by
blast
from result1 show  $(p, q) \in \Rightarrow_{nd}^* \implies$  pat-complete  $C$  (pat-mset  $p$ )  $\implies$  pat-complete
 $C$  (pat-mset  $q$ ) by blast
{
  assume  $\neg$  pat-complete  $C$  (pat-mset  $p$ )
  with wf
  show  $\exists q. (p, q) \in \Rightarrow_{nd}^! \wedge \neg$  pat-complete  $C$  (pat-mset  $q$ )
  proof (induct  $p$  rule: wf-induct[OF wf-measure[of measure-pat-poly 0]])
    case (1  $p$ )
    show ?case
    proof (cases  $p \in NF$  ( $\Rightarrow_{nd}$ ))
      case True
      thus ?thesis using 1 by auto
    next
      case False
      from nd-step-mset-pcorrect[OF False 1(2)] 1(3)
      obtain  $q$  where step:  $(p, q) \in \Rightarrow_{nd}$  and inc:  $\neg$  pat-complete  $C$  (pat-mset  $q$ )
    by auto
    from nd-step-decrease(2)[OF this(1) <improved>]
    have  $(q, p) \in$  measure (measure-pat-poly 0) by auto
    note IH = 1(1)[rule-format, OF this - inc]
    from nd-step-to-P-step[OF step, of {#}] obtain  $Q$  where
      ({#p#}, add-mset  $q$   $Q$ )  $\in \Rightarrow$  by (auto simp: P-step-def)
    from P-step-pcorrect[OF - this] 1(2) have wf-pat (pat-mset  $q$ ) by (auto
simp: wf-pats-def)
    from IH[OF this] step show ?thesis by auto
  qed
} qed
} note result2 = this
show  $\neg$  pat-complete  $C$  (pat-mset  $p$ )  $\longleftrightarrow$  ( $\exists q. (p, q) \in \Rightarrow_{nd}^! \wedge \neg$  pat-complete
 $C$  (pat-mset  $q$ ))
(is ?L = ?R)
proof
  assume ?L
  from result2[OF this] obtain  $q$  where pq:  $(p, q) \in \Rightarrow_{nd}^!$  and inc:  $\neg$  pat-complete
 $C$  (pat-mset  $q$ ) by auto
  hence  $(p, q) \in \Rightarrow_{nd}^*$  by auto
  with pq inc show ?R by auto
next

```

```

    assume ?R
    then obtain q where (p,q) ∈ ⇒nd* and ¬ pat-complete C (pat-mset q) by
    auto
    from result1[OF this(1)] this(2) show ?L by auto
  qed
end
end
end

```

7 A List-Based Implementation to Decide Pattern Completeness

```

theory Pattern-Completeness-List
  imports
    Pattern-Completeness-Multiset
    HOL-Library.AList
    HOL-Library.Mapping
    Singleton-List
begin

```

7.1 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 × 'b)list
type-synonym ('f,'v,'s)match-problem-list = (('f,nat × 's)term × ('f,'v)term)
list — mp with arbitrary pairs
type-synonym ('f,'v,'s)match-problem-lx = ((nat × 's) × ('f,'v)term) list — mp
where left components are variable
type-synonym ('f,'v,'s)match-problem-rx = ('v,('f,nat × 's)term list) alist × bool
— mp where right components are variables
type-synonym ('f,'v,'s)match-problem-fvf = ('v,(nat × 's) list) alist
type-synonym ('f,'v,'s)match-problem-lr = ('f,'v,'s)match-problem-lx × ('f,'v,'s)match-problem-rx
— a partitioned mp
type-synonym ('f,'v,'s)pat-problem-list = ('f,'v,'s)match-problem-list list
type-synonym ('f,'v,'s)pat-problem-lr = ('f,'v,'s)match-problem-lr list
type-synonym ('f,'v,'s)pat-problem-lx = ('f,'v,'s)match-problem-lx list
type-synonym ('f,'v,'s)pat-problem-fvf = ('f,'v,'s)match-problem-fvf list
type-synonym ('f,'v,'s)pat-problem-list = ('f,'v,'s)pat-problem-list list
type-synonym ('f,'v,'s)pat-problem-set-impl = (('f,nat × 's)term × ('f,'v)term)
list list

```

definition *lvars-mp* :: ('f,'v,'s)match-problem-mset ⇒ 'v set **where**
lvars-mp mp = (⋃ (vars 's nd ' mp-mset mp))

definition *vars-mp-mset* :: ('f,'v,'s)match-problem-mset ⇒ 'v multiset **where**

$vars\text{-}mp\text{-}mset\ mp = sum\text{-}mset\ (image\text{-}mset\ (vars\text{-}term\text{-}ms\ o\ snd)\ mp)$

definition $ll\text{-}mp :: ('f, 'v, 's)\text{match}\text{-}problem\text{-}mset \Rightarrow bool$ **where**
 $ll\text{-}mp\ mp = (\forall\ x.\ count\ (vars\text{-}mp\text{-}mset\ mp)\ x \leq 1)$

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

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

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

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

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

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

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

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

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

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

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

definition $pat\text{-}list :: ('f, 'v, 's)\text{pat}\text{-}problem\text{-}list \Rightarrow ('f, 'v, 's)\text{pat}\text{-}problem\text{-}set$
where $pat\text{-}list\ ps = set\ 'set\ ps$

abbreviation $pats\text{-}mset\text{-}list :: ('f, 'v, 's)\text{pats}\text{-}problem\text{-}list \Rightarrow ('f, 'v, 's)\text{pats}\text{-}problem\text{-}mset$
where $pats\text{-}mset\text{-}list \equiv mset\ o\ map\ pat\text{-}mset\text{-}list$

definition $subst\text{-}match\text{-}problem\text{-}list :: ('f, nat \times 's)\text{subst} \Rightarrow ('f, 'v, 's)\text{match}\text{-}problem\text{-}list$

$\Rightarrow ('f, 'v, 's)\text{match-problem-list}$ **where**
 $\text{subst-match-problem-list } \tau = \text{map } (\text{subst-left } \tau)$

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

definition $\text{match-var-impl} :: ('f, 'v, 's)\text{match-problem-lr} \Rightarrow 'v \text{ list} \times ('f, 'v, 's)\text{match-problem-lr}$
where

$\text{match-var-impl } mp = (\text{case } mp \text{ of } (xl, (rx, b)) \Rightarrow$
 $\text{let } xs = \text{remdups } (\text{List.maps } (\text{vars-term-list } o \text{snd}) xl)$
 $\text{in } (xs, (xl, (\text{filter } (\lambda (x, ts). \text{tl } ts \neq [] \vee x \in \text{set } xs) rx), b)))$

definition $\text{find-var} :: \text{bool} \Rightarrow ('f, 'v, 's)\text{match-problem-lr list} \Rightarrow -$ **where**
 $\text{find-var improved } p = (\text{if improved then fst } (\text{hd } (\text{List.maps } (\lambda (lx, -). \text{lx}) p)) \text{ else}$
 $\text{case List.maps } (\lambda (lx, -). \text{lx}) p \text{ of}$
 $(x, t) \# - \Rightarrow x$
 $| [] \Rightarrow \text{let } (-, rx, b) = \text{hd } p$
 $\text{in case hd } rx \text{ of } (x, s \# t \# -) \Rightarrow \text{hd } (\text{the } (\text{conflicts } s t)))$

definition $\text{empty-lr} :: ('f, 'v, 's)\text{match-problem-lr} \Rightarrow \text{bool}$ **where**
 $\text{empty-lr } mp = (\text{case } mp \text{ of } (lx, rx, -) \Rightarrow lx = [] \wedge rx = [])$

fun $\text{zipAll} :: 'a \text{ list} \Rightarrow 'b \text{ list list} \Rightarrow ('a \times 'b \text{ list}) \text{ list}$ **where**
 $\text{zipAll } [] - = []$
 $| \text{zipAll } (x \# xs) yss = (x, \text{map hd } yss) \# \text{zipAll } xs (\text{map tl } yss)$

type-synonym $('f, 's)\text{spp-list} = ('f, \text{nat} \times 's)\text{term list list list}$

datatype $('f, 'v, 's)\text{pat-impl-result} = \text{Incomplete}$
 $| \text{New-Problems } \text{nat} \times \text{nat} \times ('f, 'v, 's)\text{pat-problem-list list}$
 $| \text{Fin-Constr-Form } ('f, 's)\text{spp-list}$

Transforming finite variable forms:

definition $\text{tvars-match-list} = \text{remdups} \circ \text{concat} \circ \text{map } (\text{var-list-term} \circ \text{fst})$

definition $\text{tvars-pat-list} = \text{remdups} \circ \text{concat} \circ \text{map } \text{tvars-match-list}$

definition $\text{var-form-of-match-rx} :: ('f, 'v, 's)\text{match-problem-rx} \Rightarrow ('v \times (\text{nat} \times 's)$
 $\text{list}) \text{ list}$ **where**
 $\text{var-form-of-match-rx} = \text{map } (\text{map-prod id } (\text{map the-Var})) \circ \text{fst}$

definition $\text{match-of-var-form-list}$ **where**
 $\text{match-of-var-form-list } mpv = \text{concat } [[(\text{Var } v, \text{Var } x). v \leftarrow vs]. (x, vs) \leftarrow mpv]$

definition $\text{var-form-of-pat-rx}$ **where**
 $\text{var-form-of-pat-rx} = \text{map } \text{var-form-of-match-rx}$

definition $\text{pat-of-var-form-list}$ **where**

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

lemma *size-zip[termination-simp]*: *length ts = length ls* \implies *size-list* ($\lambda p.$ *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* \Rightarrow ('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* \Rightarrow ('f,'v,'s)*pat-problem-lx* \Rightarrow ('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 \Rightarrow *pat-inner-lin-impl p pd*
 | *Some mp'* \Rightarrow *if mp' = [] then None*
 else pat-inner-lin-impl p (mp' # pd))

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* \Rightarrow '*a* \times '*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*: ($\exists x \in \text{set } xs. \exists y \in \text{set } xs. f x \neq f y$) \longleftrightarrow

($\exists (x,y) \in \text{set } (\text{pairs-of-list } xs). f x \neq f y$) (**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) \neq f (xs ! (Suc i)) by auto*

```

from  $i$  have  $xs ! i \in set\ xs$   $xs ! (Suc\ i) \in set\ xs$  by auto
with diff show  $?l$  by auto
next
assume  $?l$ 
show  $?r$ 
proof (rule ccontr)
  let  $?n = length\ xs$ 
  assume  $\neg ?r$ 
  hence  $eq: \bigwedge i. Suc\ i < ?n \implies f\ (xs\ !\ i) = f\ (xs\ !\ (Suc\ i))$  by (auto simp: set-pairs-of-list)
  have  $eq: i < ?n \implies f\ (xs\ !\ i) = f\ (xs\ !\ 0)$  for  $i$ 
    by (induct i, insert eq, auto)
  hence  $\bigwedge i\ j. i < ?n \implies j < ?n \implies f\ (xs\ !\ i) = f\ (xs\ !\ j)$  by auto
  with  $\langle ?l \rangle$  show False unfolding set-conv-nth by auto
qed
qed

```

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

definition *compute-k-parameter* $P = max\ 2\ (Suc\ (max-list\ (map\ length\ P)))$

lemma *compute-k-parameter*: $\forall p \in set\ P. length\ p < compute-k-parameter\ P$
unfolding *compute-k-parameter-def* **using** *max-list[of - map length P]*
by *force*

lemma *compute-k-parameter-1*: *compute-k-parameter* $P > 1$
unfolding *compute-k-parameter-def* **by** *auto*

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* :: $(f,nat \times 's)term \Rightarrow 'v \Rightarrow (f,'v,'s)match-problem-rx \Rightarrow (f,'v,'s)match-problem-rx\ option$ **where**
insert-rx $t\ x\ rxb = (case\ rxb\ of\ (rx,b) \Rightarrow (case\ map-of\ rx\ x\ of$
 $None \Rightarrow Some\ (((x,[t]) \# rx,\ b)$
 $| Some\ ts \Rightarrow (case\ those\ (map\ (conflicts\ t)\ ts)$
 $of\ None \Rightarrow None\ \text{--- clash}$
 $| Some\ cs \Rightarrow if\ [] \in set\ cs\ then\ Some\ rxb\ \text{--- empty conflict means (t,x) was$
 already part of rxb
 else $Some\ ((AList.update\ x\ (t \# ts)\ rx,\ b \vee (\exists\ y \in set\ (concat\ cs). inf-sort$
 ($snd\ y))))$
 $)))$

Decomposition applies decomposition, duplicate and clash rule to classify

all remaining problems as being of kind $(x, f(l_1, \dots, l_n))$ or (t, x) .

fun *decomp-impl* :: ('f, 'v, 's)match-problem-list \Rightarrow ('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)
 \Rightarrow Some ((x, Fun g ls) # lx, rx)
| None \Rightarrow None)
| *decomp-impl* ((t, Var y) # mp) = (case *decomp-impl* mp of Some (lx, rx) \Rightarrow
(case insert-rx t y rx of Some rx' \Rightarrow Some (lx, rx') | None \Rightarrow None)
| None \Rightarrow None)

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

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

definition *match-steps-impl* :: ('f, 'v, 's)match-problem-list \Rightarrow ('v list \times ('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 \Rightarrow 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

renNat :: nat \Rightarrow 'v **and**

renVar :: 'v \Rightarrow 'v **and**

fcf-solve :: nat \Rightarrow ('f, 's)spp-list \Rightarrow bool

begin

partial-function (*tailrec*) *decomp'-main-loop* **where**
decomp'-main-loop n xs list out = (case list of
[] \Rightarrow (n, out) — one might change to (rev out) in order to preserve the order
| ((x, ts) # rxs) \Rightarrow (if tl ts = [] \vee ($\exists t \in$ set ts. is-Var t) $\vee x \in$ 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 @ rx) 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))

```

context

fixes *CC* :: 'f × 's list ⇒ 's option

begin

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 ∧ (∀ mp ∈ set no-ivc. fst mp = []) then
      Fin-Constr-Form (map (map snd o fst o snd) (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 (let x = find-var improved no-ivc; 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))))))

```

partial-function (*tailrec*) *pats-impl* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow ('f, 'v, 's)\text{pats-problem-list} \Rightarrow \text{bool}$ **where**

```

pats-impl n nl ps = (case ps of [] => True
  | p # ps1 => (case pat-impl n nl p of
    Incomplete => False
  | Fin-Constr-Form p' =>
    if fcf-solve n p' then pats-impl n nl ps1 else
    False
  | New-Problems (n', nl', ps2) => pats-impl n' nl' (ps2 @ ps1)))

```

definition *pat-complete-impl* :: $('f, 'v, 's)\text{pats-problem-list} \Rightarrow \text{bool}$ **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;
  k = compute-k-parameter ps;
  ps' = if improved then map (map (map (apsnd (map-vars renVar)))) ps else
ps
  in pats-impl n nl ps')
end
end
end

```

definition *renaming-funs* :: $(\text{nat} \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) \Rightarrow \text{bool}$ **where**

```

renaming-funs rn rx = (inj rn  $\wedge$  inj rx  $\wedge$  range rn  $\cap$  range rx = {})

```

lemmas *pat-complete-impl-code* [*code*] =
pattern-completeness-context.pat-complete-impl-def
pattern-completeness-context.pats-impl.simps
pattern-completeness-context.pat-impl-def
pattern-completeness-context. τ s-list-def
pattern-completeness-context.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

7.2 Partial Correctness of the Implementation

TODO: move

lemma *mset-sum-reindex*: $(\sum x \in \#A. \text{image-mset } (f \ x) \ B) = (\sum i \in \#B. \{\#f \ x \ i. x \in \#A\})$

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  $\bigwedge$  bs. bs  $\in$  set bss  $\implies$  length bs = n
shows zipAll as bss = map ( $\lambda$  i. (as ! i, map ( $\lambda$  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  $\wedge$  bs. bs  $\in$  set ?tbss  $\implies$  length
bs = n
  by auto
  from Cons(2-) sn have hd: bs  $\in$  set bss  $\implies$  hd bs = bs ! 0 for bs by (cases
bs) force+
  from Cons(2-) sn have tl: bs  $\in$  set bss  $\implies$  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 tvar-pat-mono:  $P \subseteq P' \implies$  tvar-pat  $P \subseteq$  tvar-pat  $P'$ 
  unfolding tvar-pat-def by auto

```

```

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

```

```

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

```

```

lemma mset-update: map-of xs  $x =$  Some  $y \implies$ 
  mset (AList.update x z xs) = (mset xs - {# (x,y) #}) + {# (x,z) #}

```

by (*induction xs, auto*)

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

lemma *mp-rx-append*: $\text{mp-rx } (xs \ @ \ ys, \ b) = \text{mp-rx } (xs,b) + \text{mp-rx } (ys,b)$
unfolding *mp-rx-def* **by** *auto*

lemma *mp-rx-Cons*: $\text{mp-rx } (p \ # \ xs, \ b) = \text{mp-list } (\text{case } p \ \text{of } (x, \ ts) \Rightarrow \text{map } (\lambda t. \\ (t, \ \text{Var } x)) \ ts) \\ + \text{mp-rx } (xs,b)$
unfolding *mp-rx-def* **by** *auto*

lemma *set-tvars-match-list*: $\text{set } (\text{tvars-match-list } mp) = \text{tvars-match } (\text{set } mp)$
by (*auto simp: tvars-match-list-def tvars-match-def*)

lemma *set-tvars-pat-list*: $\text{set } (\text{tvars-pat-list } pp) = \text{tvars-pat } (\text{pat-list } pp)$
by (*simp add: tvars-pat-list-def tvars-pat-def set-tvars-match-list pat-list-def*)

lemma *non-uniq-image-diff*: $\neg \text{UNIQ } (\alpha \ ' \ \text{set } vs) \longleftrightarrow (\exists \ v \in \ \text{set } vs. \ \exists \ w \in \ \text{set} \\ vs. \ \alpha \ v \neq \ \alpha \ w)$
by (*smt (verit, ccfv-SIG) Uniq-def image-iff*)

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

Various well-formed predicates for intermediate results

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

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

definition *wf-ts3* :: $(f, \ \text{nat} \times \ 's) \ \text{term list} \Rightarrow \ \text{bool}$ **where**
 $wf-ts3 \ ts = (\exists \ t \in \ \text{set } ts. \ \text{is-Var } t)$

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

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

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

definition $wf-rx3 :: ('f, 'v, 's)match\text{-}problem\text{-}rx \Rightarrow bool$ **where**
 $wf-rx3\ rx = (wf-rx2\ rx \wedge (improved \longrightarrow snd\ rx \vee (Ball\ (snd\ 'set\ (fst\ rx))\ wf-ts3)))$

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

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

definition $wf-lr3 :: ('f, 'v, 's)match\text{-}problem\text{-}lr \Rightarrow bool$
where $wf-lr3\ pair = (case\ pair\ of\ (lx, rx) \Rightarrow wf-lx\ lx \wedge (if\ lx = []\ then\ wf-rx3\ rx\ else\ wf-rx\ rx))$

definition $wf-pat-lr :: ('f, 'v, 's)pat\text{-}problem\text{-}lr \Rightarrow bool$ **where**
 $wf-pat-lr\ mps = (Ball\ (set\ mps)\ (\lambda\ mp.\ wf-lr3\ mp \wedge \neg\ empty-lr\ mp))$

definition $wf-pat-lx :: ('f, 'v, 's)pat\text{-}problem\text{-}lx \Rightarrow bool$ **where**
 $wf-pat-lx\ mps = (Ball\ (set\ mps)\ (\lambda\ mp.\ ll-mp\ (mp-list\ (mp-lx\ mp)) \wedge wf-lx\ mp \wedge mp \neq []))$

lemma $wf-rx-mset$: **assumes** $mset\ rx = mset\ rx'$
shows $wf-rx\ (rx, b) = wf-rx\ (rx', b)$
proof –
from $assms$ **have** $set\ rx = set\ rx'$ **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** $(auto\ simp: set)$
done

qed

lemma $wf-rx2-mset$: **assumes** $mset\ rx = mset\ rx'$
shows $wf-rx2\ (rx, b) = wf-rx2\ (rx', b)$
proof –
from $assms$ **have** $set\ rx = set\ rx'$ **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** $(auto\ simp: set)$

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 $mp-lr\ (lx,(rx,b)) = mp-lr\ (lx,(rx',b))$
using *assms* **by** (*auto simp add: mp-lr-def mp-rx-def mset-concat-union*)

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*: $(\forall j < length\ ts. \forall i < j. conflicts\ (ts\ !\ i)\ (ts\ !\ j) \neq None)$
 $\longleftrightarrow (\forall s\ t. s \in set\ ts \longrightarrow t \in set\ ts \longrightarrow conflicts\ s\ t \neq None)$ (**is** $?l = ?r$)

proof
assume $?l$
note $l = this[rule-format]$
show $?r$
proof (*intro allI impI*)
fix $s\ t$
assume $s \in set\ ts\ t \in set\ ts$
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 $(lt)\ i < j \mid (eq)\ i = j \mid (gt)\ j < i$ **by** *linarith*
thus $conflicts\ s\ t \neq None$
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'  $\implies$   $(\rightarrow_m)^{**}$  (add-mset (t, Var x) (mp-lr mp + M)) (mp-lr
(ls, rx') + M)  $\wedge$  wf-rx rx'
   $\wedge$  lvars-mp (add-mset (t, Var x) (mp-lr mp + M))  $\supseteq$  lvars-mp (mp-lr (ls, rx') +
M)
  res = None  $\implies$  match-fail (add-mset (t, Var x) (mp-lr mp + M))
proof -
  obtain rx b where rxb: rxb = (rx, b) by force
  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'
    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)  $\wedge$  wf-rx rx'
     $\wedge$  lvars-mp (mp-lr (ls, rx') + M)  $\subseteq$  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
(mpmset (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  $sx: (s, \text{Var } x) \in\# \text{ add-mset } (t, \text{Var } x) (\text{mp-rx } rxb)$  and  $y: \text{Conflict-Var } t s y$  and  $inf: \text{inf-sort } (snd y)$ 
        by blast
        from  $wf$  have  $dist: \text{distinct } (\text{map fst } rx)$  unfolding  $wf\text{-rx-def } rxb$  by
auto
      from  $y$  have  $s \neq t$  by auto
      with  $sx$  have  $(s, \text{Var } x) \in\# \text{ mp-rx } rxb$  by auto
      hence  $s \in \text{set } ts$  unfolding  $\text{mp-rx-def } rxb$  using  $\text{mem eq-key-imp-eq-value}[OF dist]$  by auto
      with  $y \text{ confl}$  have  $y \in \text{set } (\text{concat } cs)$  by (cases conflicts t s; force)
      with  $inf$  show  $?thesis$  by auto
      qed
      qed
      have  $wf: wf\text{-rx } rx'$  using  $wf \text{ ts'}$  unfolding  $wf\text{-rx-def id}$  unfolding  $rx' \text{ rxb}$ 
 $snd\text{-conv } b$  by (auto simp: distinct-update set-update[OF look])
      show  $?thesis$  using  $wf \text{ id}$  unfolding  $\text{mp}$  by (auto simp: mp-lr-def)
      qed
      qed
    }
  qed

```

lemma *decomp-impl: decomp-impl mp = res \impl*

*(res = Some mp' \impl $(\rightarrow_m)^{**} (\text{mp-list } mp + M) (\text{mp-lr } mp' + M) \wedge wf\text{-lr } mp'$*

$\wedge \text{lvars-mp } (\text{mp-list } mp + M) \supseteq \text{lvars-mp } (\text{mp-lr } mp' + M)$

*$\wedge (\text{res} = \text{None} \implies (\exists mp'. (\rightarrow_m)^{**} (\text{mp-list } mp + M) mp' \wedge \text{match-fail } mp'))$*

proof (*induct mp arbitrary: res M mp' rule: decomp-impl.induct*)

case 1

thus $?case$ **by** (*auto simp: mp-lr-def mp-rx-def wf-lr-def wf-lx-def wf-rx-def inf-var-conflict-def*)

next

case ($2 f \text{ ts } g \text{ ls } mp \text{ res } M \text{ mp}'$)

have $id: \text{mp-list } ((\text{Fun } f \text{ ts}, \text{Fun } g \text{ ls}) \# mp) + M = \text{add-mset } (\text{Fun } f \text{ ts}, \text{Fun } g \text{ ls}) (\text{mp-list } mp + M)$

by *auto*

show $?case$

proof (*cases (f,length ts) = (g,length ls)*)

case *False*

with $2(2-)$ **have** $res: res = \text{None}$ **by** *auto*

from $\text{match-clash}[OF \text{False}, \text{of } (\text{mp-list } mp + M), \text{folded } id]$

show $?thesis$ **unfolding** res **by** *blast*

next

case *True*

have $id2: \text{mp-list } (\text{zip } ts \text{ ls } @ mp) + M = \text{mp-list } mp + M + \text{mp-list } (\text{zip } ts \text{ ls})$

by *auto*

from $\text{True } 2(2-)$ **have** $res: \text{decomp-impl } (\text{zip } ts \text{ ls } @ mp) = res$ **by** *auto*

note $IH = 2(1)[OF \text{True this}, \text{of } mp' M]$

note $step = \text{match-decompose}[OF \text{True}, \text{of } \text{mp-list } mp + M, \text{folded } id \text{ id2}]$

```

    have lvars: lvars-mp (mp-list ((Fun f ts, Fun g ls) # mp) + M)  $\supseteq$  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: ( $\rightarrow_m$ )** (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)
     $\subseteq$  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: ( $\rightarrow_m$ )** (mp-lr ( lx, rx) + add-mset (t, Var y) M) (mp-lr ( lx,
rx') + M)
  and lvars: lvars-mp (mp-lr (lx, rx') + M)  $\subseteq$  lvars-mp (add-mset (t, Var y)
(mp-lr (lx, rx) + M))
  by auto
  from IH(1) steps
  have steps: ( $\rightarrow_m$ )** (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*: $\text{match-decomp-lin-impl } mp = res \implies ll\text{-mp } (mp\text{-list } mp + M) \implies$

$(res = \text{Some } mp' \longrightarrow (\rightarrow_m)^{**} (mp\text{-list } mp + M) (mp\text{-list } (mp\text{-lx } mp') + M) \wedge$
 $wf\text{-lx } mp' \wedge ll\text{-mp } (mp\text{-list } (mp\text{-lx } mp') + M))$

$\wedge (res = \text{None} \longrightarrow (\exists mp'. (\rightarrow_m)^{**} (mp\text{-list } mp + M) mp' \wedge \text{match-fail } mp'))$
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)  $\subseteq$  # 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 ...  $\rightarrow_m$  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  $\leq$  1 by auto
  hence count (vars-mp-mset (mp-list mp + M)) y = 0
  unfolding vars-mp-mset-def by auto

```

hence $y \notin \# \text{ vars-mp-mset } (mp\text{-list } mp + M)$
by (*simp add: not-in-iff*)
hence $y \notin \text{ set-mset } (\text{vars-mp-mset } (mp\text{-list } mp + M))$ **by** *blast*
also have $\text{set-mset } (\text{vars-mp-mset } (mp\text{-list } mp + M)) = \bigcup (\text{vars } ' \text{snd } ' \text{ mp-mset } (mp\text{-list } mp + M))$
unfolding *vars-mp-mset-def o-def* **by** *auto*
finally show $y \notin \bigcup (\text{vars } ' \text{snd } ' \text{ mp-mset } (mp\text{-list } mp + M))$ **by** *auto*
qed
finally have $mp\text{-list } ((t, \text{Var } y) \# mp) + M \rightarrow_m mp\text{-list } mp + M$.
note *step = converse-rtranclp-into-rtranclp[of mp-step-mset, OF this]*
note $IH = _4(1)[OF - ll\text{-new}]$
show *?case*
proof (*cases match-decomp-lin-impl mp*)
case *None*
with $IH[OF \text{None}]$ *res step* **show** *?thesis* **by** *fastforce*
next
case (*Some mp x*)
with $IH[OF \text{Some, of } mp\mathit{x}]$ *res step* **show** *?thesis* **by** *fastforce*
qed
qed

lemma *pat-inner-lin-impl: assumes pat-inner-lin-impl p pd = res*
and $wf\text{-pat-lx } pd \forall mp \in \text{set } p. ll\text{-mp } (mp\text{-list } mp)$
and $tvars\text{-pat } (\text{pat-mset } (\text{pat-mset-list } p + \text{pat-lx } pd)) \subseteq V$
shows $\text{res} = \text{None} \implies (\text{add-mset } (\text{pat-mset-list } p + \text{pat-lx } pd) P, P) \in \Rightarrow^+$
and $\text{res} = \text{Some } p' \implies (\text{add-mset } (\text{pat-mset-list } p + \text{pat-lx } pd) P, \text{add-mset } (\text{pat-lx } p') P) \in \Rightarrow^*$
 $\wedge wf\text{-pat-lx } p' \wedge tvar\text{s-pat } (\text{pat-mset } (\text{pat-lx } p')) \subseteq V$
proof (*atomize(full), insert assms, induct p arbitrary: pd res p'*)
case *Nil*
then show *?case* **by** (*auto simp: wf-pat-lr-def pat-mset-list-def pat-lr-def*)
next
case (*Cons mp p pd res p'*)
let $?p = \text{pat-mset-list } p + \text{pat-lx } pd$
have $\text{id: pat-mset-list } (mp \# p) + \text{pat-lx } pd = \text{add-mset } (mp\text{-list } mp) ?p$ **unfolding** *pat-mset-list-def* **by** *auto*
from $\text{Cons}(4)$ **have** $ll\text{-mp } (mp\text{-list } mp + \{\#\})$ **by** *auto*
note $\text{match} = \text{match-decomp-lin-impl}[OF - \text{this}]$
note $\text{res} = \text{Cons}(2)[\text{unfolded pat-inner-lin-impl.simps}]$
from $\text{Cons}(4)$ **have** $ll\text{-mp: } \forall mp \in \text{set } p. ll\text{-mp } (mp\text{-list } mp)$
and $ll\text{-mp: } ll\text{-mp } (mp\text{-list } mp)$ **by** *auto*
show *?case*
proof (*cases match-decomp-lin-impl mp*)
case (*Some mp'*)
from $\text{match}[OF \text{this, of } mp']$
have $\text{steps: } (\rightarrow_m)^{**} (mp\text{-list } mp) (mp\text{-list } (mp\text{-lx } mp'))$ **and** $wf: wf\text{-lx } mp'$
and $ll\text{-mp': } ll\text{-mp } (mp\text{-list } (mp\text{-lx } mp'))$ **by** *auto*
from $mp\text{-step-mset-steps-vars}[OF \text{steps}]$

```

have tvars: tvars-match (mp-mset (mp-list (mp-lx mp')))  $\subseteq$  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)  $\in$   $\Rightarrow$ 
      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)))  $\subseteq$  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  $\langle I2 = I1 \rangle$  by auto
  qed
next
  case None
  from match[OF None] obtain mp' where
    msteps:  $(\rightarrow_m)^{**}$  (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]
  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)  $\in$  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)  $\in$  P-step $\hat{=}$  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))  $\subseteq$  V

```

```

    using Cons(5) unfolding tvar-pat-def
    by (auto simp: pat-lx-def pat-mset-list-def)
    from IH[OF this] steps tvar
    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  $\tau$  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 - -  $\lambda$  xs.  $\sum \#$  (image-mset vars-term-ms xs)])
  (induct mp, auto)

```

```

lemma subst-conversion: map ( $\lambda$  $\tau$ . subst-pat-problem-mset  $\tau$  (pat-mset-list p)) xs
  =
  map pat-mset-list (map ( $\lambda$  $\tau$ . subst-pat-problem-list  $\tau$  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  $\tau$  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  $\tau$  p) = ll-pp p
  unfolding ll-pp-def subst-pat-problem-list-def using ll-mp-subst[of  $\tau$ ]
  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: tvar-pat (pat-list p)  $\subseteq$   $\{..<n\} \times S$ 
  and linear: ll-pp p
  shows res = None  $\implies$  (add-mset (pat-mset-list p) P, add-mset  $\{\#\}$  P)  $\in \implies^*$ 
  and res = Some ps  $\implies$  (add-mset (pat-mset-list p) P, mset (map pat-mset-list
ps) + P)  $\in \implies^+$ 
     $\wedge$  tvar-pat ( $\bigcup$  (pat-list ' set ps))  $\subseteq$   $\{..<n+m\} \times S$ 
     $\wedge$  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: tvar-pat (pat-mset (pat-mset-list p))  $\subseteq$   $\{..<n\} \times S$ 

```

```

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 lin-
ear[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)
        with 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) (τs-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) (τs-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^)) (τs-list n x) =
map pat-mset-list ps

```

```

unfolding id
  unfolding ps-def subst-pat-problem-list-def subst-pat-problem-mset-def
subst-match-problem-mset-def
  subst-match-problem-list-def map-map o-def
by (intro list.map-cong0, auto simp: pat-mset-list-def o-def image-mset.compositionality)
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  $*[unfolding\ pair]$  have  $st: (s,t) \in set\ mp'$  and  $y: (yn,\iota) \in vars\ (s \cdot \tau)$ 
  unfolding  $subst-left-def$  by auto
    from  $y[unfolding\ vars-term-subst,\ simplified]$ 
    obtain  $z$  where  $z: z \in vars\ s$  and  $y: (yn,\iota) \in vars\ (\tau\ z)$  by auto
    obtain  $f\ ss$  where  $info: info = (f,ss)$  by  $(cases\ info,\ auto)$ 
    with  $len$  have  $len: length\ ss \leq m$  by auto
    define  $ts :: ('f,-)term\ list$  where  $ts = map\ Var\ (indexed-from\ n\ ss)$ 
    from  $tau[unfolding\ \tau c-def\ info\ split]$ 
    have  $tau: \tau = subst\ x\ (Fun\ f\ ts)$  unfolding  $ts-def$  by auto
    from  $infoCl[unfolding\ Cl\ info]$ 
    have  $f: f : ss \rightarrow snd\ x\ in\ C$  by auto
    from  $C-sub-S[OF\ this]$  have  $ssS: set\ ss \subseteq S$  by simp
    from  $ssS$ 
    have  $vars\ (Fun\ f\ ts) \subseteq \{..<n + length\ ss\} \times S$ 
      by  $(auto\ simp\ add: ts-def\ in-set-indexed-from-eq)$   $(use\ nth-mem\ in\ auto)$ 
    also have  $\dots \subseteq \{..<n + m\} \times S$  using  $len$  by auto
    finally have  $subst: vars\ (Fun\ f\ ts) \subseteq \{..<n + m\} \times S$  by auto
    show  $yn \in \{..<n + m\} \wedge \iota \in S$ 
    proof  $(cases\ z = x)$ 
      case  $True$ 
        with  $y\ subst\ tau$  show  $?thesis$  by force
      next
        case  $False$ 
          hence  $\tau\ z = Var\ z$  unfolding  $tau$  by  $(auto\ simp: subst-def)$ 
          with  $y$  have  $z = (yn,\iota)$  by auto
          with  $z$  have  $y: (yn,\iota) \in vars\ s$  by auto
          with  $st$  have  $(yn,\iota) \in tvars-match\ (set\ mp')$  unfolding  $tvars-match-def$ 
    by force
    with  $mp'$  have  $(yn,\iota) \in tvars-pat\ (set\ 'set\ p'l)$  unfolding  $tvars-pat-def$ 
  by auto
    also have  $\dots = tvars-pat\ (pat-mset\ (pat-mset-list\ p'l))$ 
    by  $(rule\ arg-cong[of\ -\ tvars-pat], auto\ simp: pat-mset-list-def\ image-comp)$ 
    also have  $\dots = tvars-pat\ (pat-mset\ (pat-lx\ p'))$  unfolding  $id[symmetric]$ 
  by simp
    also have  $\dots \subseteq \{..<n\} \times S$  using  $varsp'$  .
    finally show  $?thesis$  by auto
  qed
  qed
  qed  $(insert\ res,\ auto)$ 
  qed
  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** $\forall p \in \text{set } ps. \text{tvars-pat } (\text{pat-list } p) \subseteq \{..<n\} \times S$
and *Ball* (*set ps*) *ll-pp*
and $\forall pp \in \text{pat-list } ' \text{set } ps. \text{wf-pat } pp$
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*) $\subseteq \{..<n\} \times S$
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* = (\Rightarrow) :: ((*f, 'v, 's*)*pats-problem-mset* \times (*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*]
have *steps: (pats-mset-list ps, add-mset {#} (pats-mset-list ps1))* $\in \Rightarrow^*$
by *auto*
show ?*thesis*
proof (*cases add-mset {#} (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 False*]
have (*add-mset {#} (pats-mset-list ps1), bottom-mset*) $\in \Rightarrow$ **unfolding**
P-step-def **by** *auto*
with *steps* **have** (*pats-mset-list ps, bottom-mset*) $\in \Rightarrow^*$ **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 $wf: snd \text{ ' } \bigcup (vars \text{ ' } fst \text{ ' } set (concat (concat P))) \subseteq S$
and $left-linear: Ball (set P) ll-pp$
shows $pat-complete-lin-impl (P :: ('f, 'v, 's) pats-problem-list) \longleftrightarrow pats-complete$
 $C (pat-list \text{ ' } set P)$
proof –
have $wf: Ball (pat-list \text{ ' } set P) wf-pat$
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 $n: \forall p \in set P. tvars-pat (pat-list p) \subseteq \{..<n\} \times S$
proof ($safe$)
fix $p x \iota$
assume $p: p \in set P$ **and** $xp: (x, \iota) \in tvars-pat (pat-list p)$
hence $x \in set ?l$ **unfolding** $tvars-pat-def tvars-match-def pat-list-def$
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 $\iota \in S$ **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 $(\rightarrow_m)^{**} (mp-lr mp) (mp-lr mpFin)$
and $wf-lr2 mpFin$
and $lvars-mp (mp-lr mp) \supseteq lvars-mp (mp-lr mpFin)$
and $set xs = lvars-mp (mp-list (mp-lx (fst mpFin)))$
proof –
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 $xs: xl = [] \implies xs = []$ **unfolding** $xs-def$ **by auto**
define f **where** $f = (\lambda (x, ts :: ('f, nat \times 's) term list). tl ts \neq [] \vee x \in set xs)$
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 $mp-rx-def$ **by** ($induct rx, auto$)

```

hence  $mp: mp\text{-}lr\ mp = deleted + mp'$  unfolding  $mp3\ mp\text{-}lr\text{-}def\ mp'\text{-}def\ deleted\text{-}def$ 
by auto
have  $inf\text{-}var\text{-}conflict\ (mp\text{-}mset\ (mp\text{-}rx\ (filter\ f\ rx,\ b))) = inf\text{-}var\text{-}conflict\ (mp\text{-}mset\ (mp\text{-}rx\ (rx,\ b)))$  (is  $?ivcf = ?ivc$ )
proof
  show  $?ivcf \implies ?ivc$  unfolding  $inf\text{-}var\text{-}conflict\text{-}def\ mp\text{-}rx\text{-}def\ fst\text{-}conv$  by force
  assume  $?ivc$ 
  from  $this[unfolding\ inf\text{-}var\text{-}conflict\text{-}def]$ 
  obtain  $s\ t\ x\ y$  where  $s: (s,\ Var\ x) \in\# mp\text{-}rx\ (rx,\ b)$  and  $t: (t,\ Var\ x) \in\# mp\text{-}rx\ (rx,\ b)$  and  $c: Conflict\text{-}Var\ s\ t\ y$  and  $inf: inf\text{-}sort\ (snd\ y)$ 
  by blast
  from  $c\ conflicts(5)[of\ s\ t]$  have  $st: s \neq t$  by auto
  from  $s[unfolding\ mp\text{-}rx\text{-}def]$ 
  obtain  $ss$  where  $xss: (x,ss) \in\ set\ rx$  and  $s: s \in\ set\ ss$  by auto
  from  $t[unfolding\ mp\text{-}rx\text{-}def]$ 
  obtain  $ts$  where  $xts: (x,ts) \in\ set\ rx$  and  $t: t \in\ set\ ts$  by auto
  from  $wf[unfolding\ mp3\ wf\text{-}lr\text{-}def\ wf\text{-}rx\text{-}def]$  have  $distinct\ (map\ fst\ rx)$  by auto
  from  $eq\text{-}key\text{-}imp\text{-}eq\text{-}value[OF\ this\ xss\ xts]$   $t$  have  $t: t \in\ set\ ss$  by auto
  with  $s\ st$  have  $f\ (x,ss)$  unfolding  $f\text{-}def$  by  $(cases\ ss; cases\ tl\ ss; auto)$ 
  hence  $(x,\ ss) \in\ set\ (filter\ f\ rx)$  using  $xss$  by auto
  with  $s\ t$  have  $(s,\ Var\ x) \in\# mp\text{-}rx\ (filter\ f\ rx,\ b)$   $(t,\ Var\ x) \in\# mp\text{-}rx\ (filter\ f\ rx,\ b)$ 
  unfolding  $mp\text{-}rx\text{-}def$  by auto
  with  $c\ inf$ 
  show  $?ivcf$  unfolding  $inf\text{-}var\text{-}conflict\text{-}def$  by blast
qed
also have  $\dots = b$  using  $wf$  unfolding  $mp3\ wf\text{-}lr\text{-}def\ wf\text{-}rx\text{-}def$  by auto
finally have  $ivcf: ?ivcf = b$  .
have  $wf\text{-}lr2\ ?mp'$ 
proof  $(cases\ xl = [])$ 
  case False
  from  $ivcf\ False\ wf[unfolding\ mp3]$  show  $?thesis$ 
  unfolding  $mp'\ wf\text{-}lr2\text{-}def\ wf\text{-}lr\text{-}def\ split\ wf\text{-}rx\text{-}def$  by  $(auto\ simp: distinct\text{-}map\text{-}filter)$ 
next
  case True
  with  $xs$  have  $xs = []$  by auto
  with  $True\ wf[unfolding\ mp3]$ 
  show  $?thesis$ 
  unfolding  $wf\text{-}lr2\text{-}def\ mp'\ split\ wf\text{-}rx2\text{-}def\ wf\text{-}rx\text{-}def\ ivcf$ 
  unfolding  $mp'\ wf\text{-}lr2\text{-}def\ wf\text{-}lr\text{-}def\ split\ wf\text{-}rx\text{-}def\ wf\text{-}rx2\text{-}def\ wf\text{-}ts\text{-}def\ wf\text{-}ts2\text{-}def$ 
apply  $(clarsimp\ simp: distinct\text{-}map\text{-}filter)$ 
  subgoal for  $x\ ts$  by  $(cases\ ts; cases\ tl\ ts; force)$ 
  done
qed
thus  $wf\text{-}lr2\ mpFin$  unfolding  $mpFin$  .
{
  fix  $xt\ t$ 
  assume  $del: (t,\ xt) \in\# deleted$ 

```

from *this*[*unfolded deleted-def mp-rx-def, simplified*]
obtain x ts **where** $mem: (x,ts) \in set\ rx$ **and** $nf: \neg f(x, ts)$ **and** $t: t \in set\ ts$
and $xt: xt = Var\ x$ **by force**
note $del = del[unfolded\ xt]$
from $nf[unfolded\ f-def\ split]$ t **have** $xxs: x \notin set\ xs$ **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** $xxr: x \notin fst\ 'set\ rx1 \cup fst\ 'set\ rx2$ **by auto**
define mp'' **where** $mp'' = mp-rx(filter\ (Not\ o\ f)(rx1 @ rx2), b)$
have $eq: deleted = add-mset(t, Var\ x)\ mp''$
unfolding $deleted-def\ mp''-def\ rx\ mp-rx-def\ mset-concat-union$ **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 xxr **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 <math>\neg (tl\ ts \neq [] \vee x \in set\ xs)>*) $fst-conv\ image-eqI$
prod.inject\ rx\ set-ConsD\ set-append\ ts\ xxr)
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** $yt: yt = \text{Var } y$ *add-mset pair deleted* $=$ *add-mset* $(s,$
Var $y)$ mp''
 $y \notin \bigcup (\text{vars } 'snd' (mp\text{-mset } mp'' \cup mp\text{-mset } mp'))$
by *auto*
from 1 [*unfolded yt*] **have** $y \in \bigcup (\text{vars } 'snd' (mp\text{-mset } (deleted + mp')))$
by *force*
with x **have** $x \neq y$ **by** *auto*
with *pair* yt **have** *pair* $\neq (s, \text{Var } y)$ **by** *auto*
with $yt(2)$ **have** $del: deleted = \text{add-mset } (s, \text{Var } y) (mp'' - \{\#pair\#})$
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 $lvars\text{-mp } (mp\text{-lr } mpFin) \subseteq lvars\text{-mp } (mp\text{-lr } mp)$
unfolding mp mp' *deleted-def mp'-def mpFin*
by (*auto simp: lvars-mp-def mp-lr-def*)
show $set\ xs = lvars\text{-mp } (mp\text{-list } (mp\text{-lx } (fst\ mpFin)))$
unfolding $mpFin$
unfolding $xs\text{-def lvars-mp-def mp3}$
unfolding $match\text{-var-impl-def split snd-conv fst-conv Let-def}$
by *auto*
qed

lemma *match-steps-impl: assumes match-steps-impl mp = res*
shows $res = \text{Some } (xs, mp') \implies (\rightarrow_m)^{**} (mp\text{-list } mp) (mp\text{-lr } mp') \wedge wf\text{-lr2 } mp'$
 $\wedge lvars\text{-mp } (mp\text{-list } mp) \supseteq lvars\text{-mp } (mp\text{-lr } mp')$
 $\wedge set\ xs = lvars\text{-mp } (mp\text{-list } (mp\text{-lx } (fst\ mp')))$
and $res = \text{None} \implies \exists mp'. (\rightarrow_m)^{**} (mp\text{-list } mp) mp' \wedge match\text{-fail } mp'$
proof (*atomize (full), goal-cases*)
case 1
obtain res' **where** *decomp: decomp-impl mp = res'* **by** *auto*
note $res = \text{assms}[unfolding\ match\text{-steps-impl-def decomp}]$
note $decomp = decomp\text{-impl}[OF\ decomp, of - \{\#\}, unfolding\ empty\text{-neutral}]$
show *?case*
proof (*cases res'*)
case *None*
with $decomp\ res$ **show** *?thesis* **by** *auto*
next
case (*Some mp''*)
with $decomp$ [*of mp''*]
have $steps: (\rightarrow_m)^{**} (mp\text{-list } mp) (mp\text{-lr } mp'')$ **and** $wf: wf\text{-lr } mp''$
and $lsub: lvars\text{-mp } (mp\text{-lr } mp'') \subseteq lvars\text{-mp } (mp\text{-list } mp)$ **by** *auto*
from res [*unfolded Some*] **have** $res = \text{Some } (match\text{-var-impl } mp'')$ **by** *auto*
with $match\text{-var-impl}[OF\ wf]$ $steps\ res\ lsub$ **show** *?thesis*

```

    by (cases match-var-impl mp'', auto)
  qed
qed

lemma finite-sort-imp-finite-sort-vars:
  assumes t :  $\sigma$  in  $\mathcal{T}(C, \mathcal{V})$ 
  and  $x \in \text{vars } t$ 
  and  $\neg \text{inf-sort } \sigma$ 
shows  $\neg \text{inf-sort } (\text{snd } x)$ 
  using assms
proof (induct)
  case (Fun f ts  $\sigma s$   $\sigma$ )
  from Fun obtain t where  $t \in \text{set } ts$  and  $x \in \text{vars } t$  by auto
  then obtain i where  $i : i < \text{length } ts$  and  $x : x \in \text{vars } (ts ! i)$  by (auto simp:
set-conv-nth)
  from Fun(2)[unfolded list-all2-conv-all-nth]
  have len:  $\text{length } \sigma s = \text{length } ts$  by auto
  from C-sub-S[OF Fun(1)] have inS:  $\sigma \in S$  set  $\sigma s \subseteq S$  by auto
  hence  $\sigma s : \bigwedge j. j < \text{length } ts \implies \sigma s ! j \in S$  using len unfolding set-conv-nth
by auto
show ?case
proof (rule list-all2-nthD[OF Fun(3) i, rule-format, OF x])
  show  $\neg \text{inf-sort } (\sigma s ! i)$  unfolding inf-sort[OF  $\sigma s$ [OF i]] finite-sort-def
proof
  assume inf: infinite {t. t :  $\sigma s ! i$  in  $\mathcal{T}(C)$ }
  {
    fix j
    assume  $j < \text{length } ts$ 
    from  $\sigma s$ [OF this] have  $\sigma s ! j \in S$  by auto
    from sorts-non-empty[OF this] have  $\exists tj. tj : \sigma s ! j$  in  $\mathcal{T}(C)$  by blast
  }
  hence  $\forall j. \exists tj. j < \text{length } ts \longrightarrow tj : \sigma s ! j$  in  $\mathcal{T}(C)$  by auto
  from choice[OF this] obtain tj where
     $tj : j < \text{length } ts \implies tj j : \sigma s ! j$  in  $\mathcal{T}(C)$  for j by auto
  define ft where  $ft t = \text{Fun } f (\text{map } (tj (i := t)) [0.. < \text{length } ts])$  for t
  {
    fix t
    assume  $t : \sigma s ! i$  in  $\mathcal{T}(C)$ 
    hence  $ft t : \sigma$  in  $\mathcal{T}(C)$  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 :  $\sigma s ! i$  in  $\mathcal{T}(C)$ })
  by (metis finite-imageD inj-def inj-on-def)
  with ft have infinite {t. t :  $\sigma$  in  $\mathcal{T}(C)$ }
  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 *renVar* :: 'v ⇒ 'v
and *renNat* :: nat ⇒ 'v
and *fcf-solve* :: nat ⇒ ('f,'s)spp-list ⇒ bool
and *CC* :: 'f × 's list ⇒ 's option
assumes *renaming-ass*: *improved* ⇒ *renaming-funs* *renNat* *renVar*
and *fcf-solve*: *improved* ⇒ *fcf-solver* *k* *fcf-solve*
and *CC*: *improved* ⇒ *CC* = *C*

begin

abbreviation *Match-decomp'-impl* **where** *Match-decomp'-impl* ≡ *match-decomp'-impl* *renNat*

abbreviation *Decomp'-main-loop* **where** *Decomp'-main-loop* ≡ *decomp'-main-loop* *renNat*

abbreviation *Decomp'-impl* **where** *Decomp'-impl* ≡ *decomp'-impl* *renNat*

abbreviation *Pat-inner-impl* **where** *Pat-inner-impl* ≡ *pat-inner-impl* *renNat*

abbreviation *Pat-impl* **where** *Pat-impl* ≡ *pat-impl* *renNat* *CC*

abbreviation *Pats-impl* **where** *Pats-impl* ≡ *pats-impl* *renNat* *fcf-solve* *CC*

abbreviation *Pat-complete-impl* **where** *Pat-complete-impl* ≡ *pat-complete-impl* *renNat* *renVar* *fcf-solve* *CC*

definition *allowed-vars* **where** *allowed-vars* *n* = (if *improved* then *range* *renVar* ∪ *renNat* ' {..*n*} else *UNIV*)

definition *lvar-cond* **where** *lvar-cond* *n* *V* = (*V* ⊆ *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*)

definition *size-cond-pp* **where** *size-cond-pp* *pp* = (*size* *pp* < *k*)

lemma *lvar-cond-simps*[*simp*]:

lvar-cond *n* (*insert* *x* *A*) = (*x* ∈ *allowed-vars* *n* ∧ *lvar-cond* *n* *A*)

lvar-cond *n* {}

lvar-cond *n* (*A* ∪ *B*) = (*lvar-cond* *n* *A* ∧ *lvar-cond* *n* *B*)

lvar-cond *n* (⋃ *As*) = (∀ *A* ∈ *As*. *lvar-cond* *n* *A*)

unfolding *lvar-cond-def* **by** *auto*

lemma *lvar-cond-mono*: *n* ≤ *n'* ⇒ *lvar-cond* *n* *V* ⇒ *lvar-cond* *n'* *V*

unfolding *lvar-cond-def* *allowed-vars-def* **by** (*auto* *split*: *if-splits*)

lemma *pair-fst-imageI*: (*a*,*b*) ∈ *c* ⇒ *a* ∈ *fst* ' *c* **by** *force*

lemma *not-in-fstD*: *x* ∉ *fst* ' *a* ⇒ ∀ *z*. (*x*,*z*) ∉ *a* **by** *force*

lemma *many-remdups-steps*: **assumes** *mp-mset* *mp2* = *mp-mset* *mp1* *mp2* ⊆# *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  $\bigwedge t l. (t,l) \in \# mp1 \implies \exists x. l = \text{Var } x \wedge x \notin \text{lvars-mp } (mp1 - \{(t,l) \# \} + mp2)$ 
  shows  $(\rightarrow_m)^{**}$  (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  $\notin$  lvars-mp (mp1 + mp2)
        by auto
      from match-match[of x mp1 + mp2 t, folded l, folded pair]
      have add-mset pair (mp1 + mp2)  $\rightarrow_m$  mp1 + mp2 using x unfolding lvars-mp-def
      by auto
      also have  $(\rightarrow_m)^{**}$  (mp1 + mp2) mp2
        by (rule add, insert add(2), force simp: lvars-mp-def)
      finally show ?case by simp
    qed auto
  qed

```

```

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')
   $(\rightarrow_m)^{**}$  (mp-lr mp) (mp-lr mp')
  n  $\leq$  n'
  proof (atomize (full), goal-cases)
    case 1
  end

```

```

obtain  $xl\ rx\ b$  where  $mp: mp = (xl,rx,b)$  by (cases mp, auto)
define  $out$  where  $out = (\ [] :: ('v,('f,nat \times 's)term\ list)\ alist)$ 
let  $?lr = \lambda\ rx. (xl,rx,b)$ 
define  $Measure$  where  $Measure (rx :: ('v,('f,nat \times 's)term\ list)\ alist) =$ 
   $sum\ list\ (map\ ((\lambda\ ts. sum\ list\ (map\ size\ ts))\ o\ snd)\ rx)$  for  $rx$ 
define  $cond3$  where  $cond3\ ts = (xl = [] \longrightarrow wf\ ts3\ ts)$  for  $ts$ 
{
  fix  $out\ rx'$ 
  assume  $Decomp'\text{-main-loop}\ n\ xs\ rx\ out = (n',\ rx')$ 
   $wf\ lr2\ (?lr\ (rx\ @\ out))$ 
   $lvar\ cond\ mp\ n\ (mp\ lr\ (?lr\ (rx\ @\ out)))$ 
   $Ball\ (snd\ 'set\ out)\ cond3$ 
  hence  $wf\ lr3\ (?lr\ rx') \wedge lvar\ cond\ mp\ n'\ (mp\ lr\ (?lr\ rx'))$ 
   $\wedge (\rightarrow_m)^{*}\ (mp\ lr\ (?lr\ (rx\ @\ out)))\ (mp\ lr\ (?lr\ rx'))$ 
   $\wedge n \leq n'$ 
  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  $Decomp'\text{-main-loop}\ n\ xs\ rx\ out = (n',\ rx')$  by fact
  note  $res = this[unfolded\ decomp'\text{-main-loop.simps}[of\ -\ -\ -\ rx]]$ 
  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  $?cond = tl\ ts = [] \vee (\exists\ t \in set\ ts. is\ Var\ t) \vee x \in set\ xs$ 
  note  $res = res[unfolded\ rx\ split\ list.simps]$ 
  from  $wf[unfolded\ rx\ wf\ lr2\ def\ split]$  have  $wfts: wf\ ts\ ts \vee wf\ ts2\ ts$ 
  and  $dist\ vars: distinct\ (x\ \# map\ fst\ (rx2\ @\ out))$ 
  by (auto simp: wf\ rx\ def\ wf\ rx2\ def\ split: if\ splits)
  hence  $ts: ts \neq []$  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 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 (λ i. 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 (λ j. (fresh ! j, map (λ bs. bs ! j) (map args ts))) [0..<l]
by auto
also have ... = map (λ i. (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 (λ i. (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 $?meas\ frrx \leq ?meas\ rrx$ **unfolding** *frrx-def* **by** (*induct rrx , auto*)
also have $\dots \leq ?meas\ rx1$ **unfolding** *rrx-def*
by (*induct $rx1$, auto simp: o-def split: prod.splits intro!: add-mono sum-remdups*)
also have $\dots = (\sum x \leftarrow [0..<l]. \sum xa \leftarrow [0..<k]. size\ (t\ xa\ x))$
unfolding *rx1 map-map o-def snd-conv* **by** *simp*
also have $\dots = (\sum xa \leftarrow [0..<k]. \sum x \leftarrow [0..<l]. size\ (t\ xa\ x))$
unfolding *sum.list-conv-set-nth* **by** (*auto intro: sum.swap*)
also have $\dots < sum-list\ (map\ size\ ts)$
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:*)
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:*)
have cong: $mp0 + mp2 \rightarrow_m mp1 + mp2 \implies mp1 = mp1' \implies mp0 + mp2 \rightarrow_m mp1' + mp2$
for $mp0\ mp2\ mp1\ mp1' :: ('f, 'v, 's)match-problem-mset$ **by** *auto*
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 $lvars-fresh-disj: lvars-mp\ (mp-lr\ (xl, rx\ @\ out, b)) \cap set\ fresh = \{\}$
proof –
have $lvars-mp\ (mp-lr\ (xl, rx\ @\ out, b)) \subseteq allowed-vars\ n$
using *1(4) unfolding lvar-cond-mp-def lvar-cond-def* .
moreover have $set\ fresh \cap allowed-vars\ n = \{\}$ **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: $mp-lr\ (xl, rx\ @\ out, b) \rightarrow_m mp-lr\ (xl, (rx1\ @\ rx2)\ @\ out, b)$
unfolding *left right*
proof (*rule cong[OF match-decompose'[OF - - - lfresh - <improved>, of - x f]]*)
show $(ti, y) \in\# mp-rx\ ([x, ts], b) \implies y = Var\ x \wedge root\ ti = Some\ (f, l)$ **for** $ti\ y$
unfolding *ts-t mp-rx-def* **by** *auto*
from *False* **have** $xs: x \notin set\ xs$ **by** *auto*
show $(ti, y) \in\# mp-lr\ (xl, rx2\ @\ out, b) \implies x \notin vars\ y$ **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:  $\bigcup$  (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))  $\subseteq$  allowed-vars n
    using 1(4) unfolding lvar-cond-mp-def lvar-cond-def .
    moreover have set fresh  $\cap$  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))  $\cap$  set fresh = {}
by auto
qed
show  $2 \leq \text{size} (\text{mp-rx} (\mathbb{[(x, ts)]}, b))$ 
  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 ( $\lambda$  i. Var (fresh ! i)) [0..l]
  unfolding lfresh[symmetric]
  by (intro nth-equalityI, auto)
  have ( $\sum (t, l) \in \# \text{mp-rx} (\mathbb{[(x, ts)]}, b). \text{mp-list} (\text{zip} (\text{args } t) (\text{map } \text{Var } \text{fresh}))$ )
  = mset (concat (map ( $\lambda$  t. zip (args t) (map Var fresh)) ts))
  unfolding mp-rx-def by (induct ts, auto)
  also have ... = mset (concat (map ( $\lambda$  j. map ( $\lambda$  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 ( $\sum (t, l) \in \# \text{mp-rx} (\mathbb{[(x, ts)]}, b). \text{mp-list} (\text{zip} (\text{args } t) (\text{map } \text{Var } \text{fresh}))$ ) =
  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))  $\subseteq$  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))  $\subseteq$  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  $\in$  lvars-mp (mp-lr (xl, (rx1 @ rx2) @ out, b))
  hence y  $\in$  lvars-mp (mp-lr (xl, rx @ out, b))  $\vee$  y  $\in$  lvars-mp (mp-rx
(rx1,b))
    unfolding rx'
    unfolding mp-lr-def split lvars-mp-def mp-rx-append by auto
    thus y  $\in$  allowed-vars (n + l)
  proof
    assume y  $\in$  lvars-mp (mp-lr (xl, rx @ out, b))
    with lvc have y  $\in$  allowed-vars n unfolding lvar-cond-mp-def
lvar-cond-def by auto
    thus ?thesis unfolding allowed-vars-def by auto
  next
    assume y  $\in$  lvars-mp (mp-rx (rx1, b))
    hence y  $\in$  set fresh unfolding rx1 lvars-mp-def mp-rx-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))  $\subseteq$  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-fscd: distinct (map fst (rx @ out)) by (auto split: if-splits)
have dist-mid: distinct (map fst ((rx1 @ rx2) @ out))
proof –

```

```

from dist-fscd 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 $frrx\text{-sub}\text{-}rx1: mp\text{-mset } (mp\text{-rx } (frrx, b)) \subseteq mp\text{-mset } (mp\text{-rx } (rx1, b))$.

hence $new\text{-sub}\text{-}mid: mp\text{-mset } ?new \subseteq mp\text{-mset } ?mid$ **unfolding** $mp\text{-rx}\text{-append}$
by *auto*

have $b\text{-correct}: (b = inf\text{-var}\text{-conflict } (mp\text{-mset } ?new)) = True$

proof –

let $?old = mp\text{-rx } (rx @ out, b)$

let $?mid = mp\text{-rx } ((rx1 @ rx2) @ out, b)$

let $?new = mp\text{-rx } ((frrx @ rx2) @ out, b)$

from $wf[unfolding\ wf\text{-lr}2\text{-def } wf\text{-rx}2\text{-def } wf\text{-rx}\text{-def}]$

have $b = inf\text{-var}\text{-conflict } (mp\text{-mset } ?old)$ **by** (*auto split: if-splits*)

also have $\dots = inf\text{-var}\text{-conflict } (mp\text{-mset } ?new)$ (**is** $?inf\text{-fscd} = ?inf\text{-new}$)

proof

assume $?inf\text{-fscd}$

from $this[unfolding\ inf\text{-var}\text{-conflict}\text{-def}]$

obtain $u\ w\ y\ z$ **where**

$u: (u, Var\ y) \in\# ?old$ **and**

$w: (w, Var\ y) \in\# ?old$ **and**

$conf: Conflict\text{-}Var\ u\ w\ z$ **and**

$inf: inf\text{-sort } (snd\ z)$ **by** *auto*

show $?inf\text{-new}$

proof (*cases* $y = x$)

case *False*

hence $(u, Var\ y) \in\# ?new$ $(w, Var\ y) \in\# ?new$ **using** $u\ w$

unfolding $rx\ mp\text{-rx}\text{-append}\ mp\text{-rx}\text{-Cons}\ split$ **by** *auto*

with $conf\ inf$ **show** $?thesis$ **unfolding** $inf\text{-var}\text{-conflict}\text{-def}$ **by** *blast*

next

case *True*

with $dist\text{-fscd}\ u\ w$ **have** $uw\text{-ts}: u \in set\ ts\ w \in set\ ts$

unfolding $rx\ mp\text{-rx}\text{-Cons}\ mp\text{-rx}\text{-append}\ split$

by (*auto simp: mp-rx-def dest!: not-in-fstD*)

with $conf$ **have** $uw: u \neq w$ **by** *auto*

from $uw\text{-ts}(1)$ **obtain** i **where** $i: i < k$ **and** $u: u = ts ! i$

unfolding $k\text{-def}$ **by** (*auto simp: set-conv-nth*)

from $uw\text{-ts}(2)$ **obtain** j **where** $j: j < k$ **and** $w: w = ts ! j$

unfolding $k\text{-def}$ **by** (*auto simp: set-conv-nth*)

from $u\ w\ uw$ **have** $ij: i \neq j$ **by** *auto*

have $id: ((f, length [0..<l]) = (f, length [0..<l])) = True$ **by** *simp*

have $Conflict\text{-}Var\ (Fun\ f\ (map\ (t\ i)\ [0..<l]))\ (Fun\ f\ (map\ (t\ j)\$

$[0..<l]))\ z$

using $conf[unfolding\ u\ w\ ts\ t]\ i\ j$ **by** *auto*

note $*$ = $this[unfolding\ conflicts.\text{simps}\ length\text{-map}\ id\ if\text{-True}]$

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 $*[unfolding\ those]$ **obtain** c **where** $c: c \in set\ cs$ **and** $z: z \in set$

c **by** *auto*

from $arg\text{-cong}[OF\ those[unfolding\ those\ eq\ Some],\ of\ length]$

```

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 (λj. 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 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-fscd
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..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..by auto
    then obtain css where those: those (map2 conflicts (map (t i)
[0..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 lcscs: length css = l by auto
    from arg-cong[OF those[unfolded those-eq-Some], of λ xs. xs ! a] lcscs
conf z
      have z ∈ set (css ! a) using a by simp
      with lcscs 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
} 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**

fact **have** mp-lr (xl, rx @ out, b) →_m mp-lr (xl, (rx1 @ rx2) @ out, b) **by**

```

also have ( $\rightarrow_m$ )** ( $mp\text{-}lr$  ( $xl$ , ( $rx1$  @  $rx2$ ) @  $out$ ,  $b$ )) ( $mp\text{-}lr$  ( $xl$ , ( $rrx$  @
 $rx2$ ) @  $out$ ,  $b$ ))
proof (rule many-remdups-steps[OF glob-rrx-set-eq[symmetric]])
  have  $mp\text{-}rx$  ( $rrx$ ,  $b$ )  $\subseteq\#$   $mp\text{-}rx$  ( $rx1$ ,  $b$ )
  unfolding  $rrx\text{-}def$   $mp\text{-}rx\text{-}def$   $fst\text{-}conv$ 
  proof (induct rx1)
    case (Cons pair rx2)
    obtain  $x$   $ts$  where  $pair$ :  $pair = (x,ts)$  by force
    have  $\langle\#\{t, Var\ x\}. t \in\#$   $mset$  ( $remdups\ ts\#\}$  $\rangle \subseteq\#$   $\langle\#\{t, Var\ x\}. t$ 
 $\in\#$   $mset\ ts\#\}$  $\rangle$ 
      using  $mset\text{-}remdups\text{-}subset\text{-}eq$  by (rule image-mset-subseteq-mono)
    then show ?case
      using  $subset\text{-}mset.add\text{-}mono$  [OF - Cons, of  $\langle\#\{t, Var\ x\}. t \in\#$ 
 $mset$  ( $remdups\ ts\#\}$  $\rangle$   $\langle\#\{t, Var\ x\}. t \in\#$   $mset\ ts\#\}$  $\rangle$ ]
      by (auto simp add: pair)
    qed auto
    thus  $mp\text{-}lr$  ( $xl$ , ( $rrx$  @  $rx2$ ) @  $out$ ,  $b$ )  $\subseteq\#$   $mp\text{-}lr$  ( $xl$ , ( $rx1$  @  $rx2$ ) @
 $out$ ,  $b$ )
    unfolding  $mp\text{-}lr\text{-}def$   $split$   $mp\text{-}rx\text{-}append$  by auto
  qed

also have ( $\rightarrow_m$ )** ( $mp\text{-}lr$  ( $xl$ , ( $rrx$  @  $rx2$ ) @  $out$ ,  $b$ )) ( $mp\text{-}lr$  ( $xl$ , ( $frrx$ 
@  $rx2$ ) @  $out$ ,  $b$ ))
proof -
  define  $long$  :: ( $'v \times ('f, nat \times 's)$   $Term.term$  list)  $\Rightarrow$   $bool$  where
     $long = (\lambda(y, ts'). tl\ ts' \neq [])$ 
  define  $short$  where  $short = Not\ o\ long$ 
  have  $short\text{-}long$ :  $mp\text{-}rx$  ( $rrx, b$ ) =  $mp\text{-}rx$  ( $filter\ short\ rrx, b$ ) +  $mp\text{-}rx$ 
( $filter\ long\ rrx, b$ )
  unfolding  $mp\text{-}rx\text{-}def$   $fst\text{-}conv$   $short\text{-}def$  by (induct rrx, auto)
  hence  $expand$ :  $mp\text{-}lr$  ( $xl$ , ( $rrx$  @  $rx2$ ) @  $out$ ,  $b$ ) =
 $mp\text{-}rx$  ( $filter\ short\ rrx, b$ ) +  $mp\text{-}lr$  ( $xl$ , ( $frrx$  @  $rx2$ ) @  $out$ ,  $b$ )
  unfolding  $mp\text{-}lr\text{-}def$   $split$   $mp\text{-}rx\text{-}append$   $short\text{-}long$   $long\text{-}def$   $frrx\text{-}def$ 
by simp
  show ?thesis unfolding  $expand$ 
  proof (rule many-match-steps)
    fix  $s$   $lhs$ 
    assume ( $s, lhs$ )  $\in\#$   $mp\text{-}rx$  ( $filter\ short\ rrx, b$ )
    from  $this$  [unfolded mp-rx-def fst-conv short-def long-def, simplified]
    obtain  $y$   $ts'$  where  $in\text{-}rrx$ : ( $y, ts'$ )  $\in$   $set\ rrx$  and  $lhs$ :  $lhs = Var\ y$ 
and  $tl\ ts' = []$   $s \in set\ ts'$ 
    by auto
    then have  $ts'$ :  $ts' = [s]$  by (cases ts'; cases tl ts'; auto)
    show  $\exists x. lhs = Var\ x \wedge x \notin$   $lvars\text{-}mp$ 
      ( $mp\text{-}rx$  ( $filter\ short\ rrx, b$ ) -  $\{\#\{s, lhs\}\#\}$ ) +  $mp\text{-}lr$  ( $xl$ , ( $frrx$  @
 $rx2$ ) @  $out$ ,  $b$ ))
    proof (intro exI[of -] conjI lhs notI)
      assume  $mem$ :  $y \in$   $lvars\text{-}mp$  ( $mp\text{-}rx$  ( $filter\ short\ rrx, b$ ) -  $\{\#\{s,$ 
 $lhs\}\#\}$ ) +  $mp\text{-}lr$  ( $xl$ , ( $frrx$  @  $rx2$ ) @  $out$ ,  $b$ ))

```

from *in-rrx* **obtain** *a*
where *a*: $a < l$ **and** *y*: $y = \text{fresh } ! a$ **and** *yts'*: $(y, ts') = rrx ! a$
unfolding *rrx-def rx1* **by** *auto*
with *lfresh* **have** *yfresh*: $y \in \text{set fresh}$ **by** *auto*
with *lwars-fresh-disj* *mem*
have $y \in \text{lwars-mp } (mp\text{-rx } (\text{filter short } rrx, b) - \{\#(s, lhs)\#}) \vee y \in \text{lwars-mp } (mp\text{-rx } (frrx, b))$
unfolding *rx mp-lr-def split mp-rx-append mp-rx-Cons lwars-mp-def*
by *auto*
hence $\exists b. b < l \wedge y = \text{fresh } ! b \wedge a \neq b$
proof
assume $y \in \text{lwars-mp } (mp\text{-rx } (frrx, b))$
from *this[unfolded frrx-def, folded long-def, unfolded mp-rx-def lwars-mp-def, simplified]*
obtain *ts''* **where** *ts''*: $(y, ts'') \in \text{set } rrx$ **and** *long*: *long* (y, ts'')
by *auto*
from *long ts'* **have** *diff*: $ts' \neq ts''$ **unfolding** *long-def* **by** *auto*
from *ts''* **obtain** *b* **where** $b < l$ **and** *yts''*: $(y, ts'') = rrx ! b$
and *y*: $y = \text{fresh } ! b$
unfolding *rrx-def rx1* **by** *auto*
from *yts' yts'' diff* **have** *diff*: $a \neq b$
by *(metis snd-conv)*
with *a b y* **show** *?thesis* **by** *auto*
next
assume *mem*: $y \in \text{lwars-mp } (mp\text{-rx } (\text{filter short } rrx, b) - \{\#(s, lhs)\#})$
define *other* **where** *other* = *take a rrx @ drop (Suc a) rrx*
have *lenrrx*: *length rrx = l* **unfolding** *rrx-def rx1* **by** *auto*
hence $rrx = \text{take } a \text{ rrx } @ rrx ! a \# \text{drop } (Suc \ a) \text{ rrx}$ **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** $\dots = \text{filter short } (take \ a \ rrx) \ @ \ rrx \ ! \ a \ \# \ \text{filter short } (drop \ (Suc \ a) \ rrx)$
(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** $mp\text{-rx } (?f1 \ @ \ \dots \ \# \ ?f2, \ b) - \{\#(s, lhs)\#} = mp\text{-rx } (?f1 \ @ \ ?f2, \ b)$
unfolding *mp-rx-append mp-rx-Cons lhs split* **by** *auto*
finally **have** $y \in \text{lwars-mp } (mp\text{-rx } (?f1 \ @ \ ?f2, \ b))$ **using** *mem* **by** *auto*
from *this[unfolded lwars-mp-def mp-rx-def, folded filter-append, folded other-def]*
obtain *ts''* **where** $(y, ts'') \in \text{set } other$ **by** *auto*
also **have** $\dots \subseteq \{rrx ! b \mid b. b \in \{..<length \ rrx\} - \{a\}\}$ **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  $cond: improved \implies b \vee xl \neq []$  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  $steps12: (\rightarrow_m)^{**}\ (mp-list\ mp)\ (mp-lr\ mp2)$ 
      and  $wf2: wf-lr2\ mp2$ 
      and  $xs: set\ xs = lvars-mp\ (mp-list\ (mp-lx\ (fst\ mp2)))$  by  $auto$ 
    from  $decomp'-impl[OF\ wf2\ xs\ lvc\ dec\ \langle improved \rangle]$   $steps12$ 
    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  $tvars-pat\ (pat-mset\ (pat-mset-list\ p + pat-lr\ pd)) \subseteq V$ 
  and  $lvar-cond-pp\ n\ (pat-mset-list\ p + pat-lr\ pd) \wedge size-cond-pp\ (pat-mset-list\ p$ 
 $+ pat-lr\ pd)$ 
  shows  $res = None \implies (add-mset\ (pat-mset-list\ p + pat-lr\ pd)\ P, P) \in \Rightarrow^+$ 
  and  $res = Some\ (n', p') \implies (add-mset\ (pat-mset-list\ p + pat-lr\ pd)\ P, add-mset$ 
 $(pat-lr\ p')\ P) \in \Rightarrow^*$ 
     $\wedge wf-pat-lr\ p' \wedge tvars-pat\ (pat-mset\ (pat-lr\ p')) \subseteq V$ 
     $\wedge lvar-cond-pp\ n'\ (pat-lr\ p') \wedge size-cond-pp\ (pat-lr\ p') \wedge n \leq n'$ 
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  $id: pat-mset-list\ (mp\ \# p) + pat-lr\ pd = add-mset\ (mp-list\ mp)\ ?p$  unfolding
 $pat-mset-list-def$  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)  $\wedge$  size-cond-pp
    (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))  $\wedge$ 
  size-cond-pp (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 size-cond-pp-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)
    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+ 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)  $\wedge$  size-cond-pp (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 size-cond-pp-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)  $\subseteq$   $\{..<n\} \times S$ 
  and lvarsAll:  $\forall$  pp  $\in$  # add-mset (pat-mset-list p)  $P$ . lvar-cond-pp nl pp  $\wedge$ 
size-cond-pp pp
  shows res = Incomplete  $\implies$  (add-mset (pat-mset-list p)  $P$ , add-mset {#}  $P$ )  $\in$ 
 $\implies^*$ 
  and res = New-Problems (n',nl',ps)  $\implies$  (add-mset (pat-mset-list p)  $P$ , mset
(map pat-mset-list ps) +  $P$ )  $\in$   $\implies^+$ 
   $\wedge$  tvars-pat ( $\bigcup$  (pat-list ' set ps))  $\subseteq$   $\{..<n'\} \times S$ 
   $\wedge$  ( $\forall$  pp  $\in$  # mset (map pat-mset-list ps) +  $P$ . lvar-cond-pp nl' pp  $\wedge$ 
size-cond-pp pp)  $\wedge$  n  $\leq$  n'
  and res = Fin-Constr-Form fcf  $\implies$  improved  $\wedge$  ( $\exists$  P'. (add-mset (pat-mset-list
p)  $P$ , add-mset P'  $P$ )  $\in$   $\implies^*$ 
   $\wedge$  finite-constructor-form-pat (set3 fcf)
   $\wedge$  tvars-spat (set3 fcf)  $\subseteq$   $\{..<n\} \times S$ 
   $\wedge$  length fcf  $<$  k
   $\wedge$  pat-complete C (pat-mset P') = simple-pat-complete C SS (set3 fcf))

```

proof (atomize(full), goal-cases)

```

case 1
let ?Pat-inner-impl = pat-inner-impl renNat
have wf: wf-pat-lr [] unfolding wf-pat-lr-def by auto
have vars: tvars-pat (pat-mset (pat-mset-list p))  $\subseteq$   $\{..<n\} \times S$ 
  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)  $\wedge$  size-cond-pp (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') ∧ size-cond-pp (pat-lr p') and nl: nl
≤ nl'' by auto
obtain ivc no-ivc where part: partition (λmp. 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
  thus ?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 parti-
tion-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

show ?thesis
proof (cases improved ∧ (∀ mp∈set no-ivc. fst mp = []))
case False
hence id: (improved ∧ (∀ mp∈set no-ivc. fst mp = [])) = False by auto
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
note res = res[unfolded id if-False]
define p'l where p'l = map mp-lr-list p'
define x where x = find-var improved no-ivc
define ps where ps = map (λτ. subst-pat-problem-list τ p'l) (τs-list n x)
  have id: pat-lr p' = pat-mset-list p'l unfolding p'l-def by (simp add:
pat-mset-list-lr)
  have subst: map (λτ. subst-pat-problem-mset τ (pat-lr p')) (τs-list n x) =
map pat-mset-list ps
  unfolding id
  unfolding ps-def subst-pat-problem-list-def subst-pat-problem-mset-def
subst-match-problem-mset-def
  subst-match-problem-list-def map-map o-def
  by (intro list.map-cong0, auto simp: pat-mset-list-def o-def image-mset.compositionality)
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 x-def 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 = x-def[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
      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
      hence ( $\forall mp \in set\ no-ivc. fst\ mp = []$ ) by auto
      with False have impr: ¬ improved and id: improved = False by auto
      note x = x[unfolded id if-False Nil list.simps]
      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
      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:  $\neg$  inf-var-conflict (mp-mset
(mp-rx (rx, False)))
  and wf: wf-ts2 ts unfolding rx by auto
from wf[unfolded wf-ts2-def] obtain s t ts' where ts: ts = s # t # ts'
and
  diff: s  $\neq$  t and conf: conflicts s t  $\neq$  None
  by (cases ts; cases tl ts, auto)
  from conf obtain xs where conf: conflicts s t = Some xs by (cases
conflicts s t, auto)
  with conflicts(5)[of s t] diff have xs  $\neq$  [] by auto
  with x[unfolded 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 conf: Conflict-Var s t x by auto
  from ts rx have sty: (s, Var y)  $\in$ # mp-rx (rx, False) (t, Var y)  $\in$ #
mp-rx (rx, False)
  by (auto simp: mp-rx-def)
  with conf ninf have  $\neg$  inf-sort (snd x) unfolding inf-var-conflict-def
by blast
  with sty conf rx have main:  $\neg$  improved  $\wedge$  (s, Var y)  $\in$ # mp-lr mp  $\wedge$ 
(t, Var y)  $\in$ # mp-lr mp  $\wedge$  Conflict-Var s t x  $\wedge$   $\neg$  inf-sort (snd x)
  using False impr
  unfolding mp by (auto simp: mp-lr-def)
  from mpp obtain p'' where pat: pat-lr p' = add-mset (mp-lr mp) p''
  unfolding pat-lr-def by simp (metis in-map-mset mset-add set-mset-mset)
  from pat-instantiate[OF - disjI2[OF main], of n p'', folded pat, OF disj]
  show ?thesis unfolding subst .
qed
qed
have tvars: tvars-pat ( $\bigcup$  (pat-list ' set ps))  $\subseteq$   $\{..<n + m\} \times S$ 
proof (safe del: conjI)
  fix yn ι
  assume (yn,ι)  $\in$  tvars-pat ( $\bigcup$  (pat-list ' set ps))
  then obtain pi mp
    where pi: pi  $\in$  set ps
    and mp: mp  $\in$  set pi and y: (yn,ι)  $\in$  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  $\tau$  where tau:  $\tau \in$  set ( $\tau$ -list n x) and pi: pi = map (map (subst-left
 $\tau$ )) p'l by auto
  from tau[unfolded  $\tau$ -list-def]
  obtain info where infoCl: info  $\in$  set (Cl (snd x)) and tau:  $\tau = \tau c n x$ 
info by auto
  from Cl-len[of snd x] this(1) have len: length (snd info)  $\leq$  m by force
  from mp[unfolded pi set-map] obtain mp' where mp': mp'  $\in$  set p'l and
mp: mp = map (subst-left  $\tau$ ) 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 (indexed-from n 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) ⊆ {.. $n + \text{length } ss$ } × S
      by (auto simp add: ts-def in-set-indexed-from-eq) (use nth-mem in auto)
    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 ∧ size-cond-pp pp
    proof (cases pp ∈# P)
      case True
        with lvarsAll have lvar-cond-pp nl pp size-cond-pp 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 id1: 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)
      have id2: size-cond-pp pp = size-cond-pp (pat-lr p')
        unfolding pp pp' id
        unfolding size-cond-pp-def
        by (force simp: subst-pat-problem-list-def subst-match-problem-list-def
subst-left-def pat-mset-list-def)
      show ?thesis using lvar-p' id1 id2 unfolding lvar-cond-pp-def by auto
    qed
  }
with tvars step steps res nl show ?thesis by auto
next
case all-lhs-Var: True
hence (improved ∧ (∀ mp ∈ set no-ivc. fst mp = [])) = True by auto
note res = res[unfolded this if-True]
from all-lhs-Var have impr: improved by auto
note res = res[unfolded CC[OF impr]]
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 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 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)

have no-inf-sort:  $\forall x \in \text{tvars-pat} (\text{pat-mset} (\text{pat-lr } P')). \neg \text{inf-sort} (\text{snd } x)$ 
proof
  fix y
  assume  $y \in \text{tvars-pat} (\text{pat-mset} (\text{pat-lr } P'))$ 
  from this[unfolded tvars-pat-def pat-lr-def, simplified] obtain mp
    where mp:  $mp \in \text{set } P'$  and  $y: y \in \text{tvars-match} (\text{mp-mset} (\text{mp-lr } mp))$ 
    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:  $mp \in \text{set no-ivc}$  and fmp:  $?f \text{ mp}$  by
auto
  from no-ivc-b[OF mp] all-lhs-Var mp
  obtain rx where mp-id:  $mp = ([], rx, \text{False})$ 
    and ninf:  $\neg \text{inf-var-conflict} (\text{mp-mset} (\text{mp-rx} (rx, \text{False})))$ 
    by (cases mp, auto)
  note fmp = fmp[unfolded mp-id snd-conv fst-conv]
  have id:  $mp\text{-lr } mp = \text{mp-rx} (rx, \text{False})$  unfolding mp-id mp-lr-def by auto
  from y[unfolded id mp-rx-def tvars-match-def]
  obtain x ts t where xts:  $(x, ts) \in \text{set } rx$  and t:  $t \in \text{set } ts$  and  $y: y \in \text{vars}$ 
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 impr
  have wf-ts3 ts and wf2: wf-rx2 (rx, False) by auto
  from this[unfolded wf-ts3-def] obtain z where  $z: \text{Var } z \in \text{set } ts$  by auto
  have sort:  $\mathcal{T}(C, \mathcal{V}) (\text{Var } z) = \text{Some} (\text{snd } z)$  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  $s: s \in \text{set } ts$  and  $sz: s$ 
 $\neq \text{Var } z$ 
    by (cases ts; cases tl ts, auto)
  from wf2[unfolded wf-ts2-def wf-ts-no-conflict-alt-def]
  have no-conf:  $s \in \text{set } ts \implies t \in \text{set } ts \implies \text{conflicts } s \ t \neq \text{None}$  for  $s \ t$ 
by auto
  from s z xts have
    mem:  $(\text{Var } z, \text{Var } x) \in \text{mp-mset} (\text{mp-rx} (rx, \text{False}))$ 
    (s, Var x)  $\in \text{mp-mset} (\text{mp-rx} (rx, \text{False}))$ 
    unfolding mp-rx-def by auto
  from no-conf[OF z s]
  have Conflict-Var (Var z) s z using sz by (cases s, auto simp: con-
flicts.simps)

```

with *ninf mem* **have** *ninf* : \neg *inf-sort* (*snd z*)
unfolding *inf-var-conflict-def* **by** *blast*
define σ **where** $\sigma = \text{snd } z$
from *single t*
have $t : t : \sigma$ **in** $\mathcal{T}(C, \mathcal{V})$ **unfolding** *hastype-def* σ -*def* **by** *auto*
from $t y$ *ninf*[*folded* σ -*def*]
show \neg *inf-sort* (*snd y*)
by (*rule finite-sort-imp-finite-sort-vars*)
qed

from *no-ivc-filter* **have** *set no-ivc* \subseteq *set p'* **by** *auto*
hence *steps2*: (*add-mset* ($M + \text{pat-lr no-ivc}$) *P*, *add-mset* ($M + \text{pat-lr } P'$)
P) $\in \Rightarrow^*$ **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* ($M + \text{pat-lr } (mp \# mps)$) *P* = *add-mset* (($M + \text{pat-lr}$
 $[\text{mp}]) + \text{pat-lr } mps$) *P*
unfolding *pat-lr-def* **by** *auto*
also have (\dots , *add-mset* (($M + \text{pat-lr } [\text{mp}]) + \text{pat-lr } (\text{filter } ?f mps)$) *P*)
 $\in \Rightarrow^*$
by (*rule Cons(1)*, *insert Cons*, *auto*)
also have ($M + \text{pat-lr } [\text{mp}]) + \text{pat-lr } (\text{filter } ?f mps)$ = $M + \text{pat-lr } (\text{filter}$
 $?f (mp \# mps))$
unfolding *pat-lr-def* **using** *True* **by** *auto*
finally show *?thesis* .
next
case *False*
have *add-mset* ($M + \text{pat-lr } (mp \# mps)$) *P* = *add-mset* (*add-mset*
 $(\text{mp-lr } mp) (M + \text{pat-lr } mps)$) *P*
unfolding *pat-lr-def* **by** *simp*
also have (\dots , $\{\# M + \text{pat-lr } mps \# \} + P$) $\in \Rightarrow$ **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*: $(xl, xr, b) \in \text{set } p'$ **by** *auto*
from *mp False* **obtain** $x \ ts$ **where** $xts : (x, ts) \in \text{set } xr$
and *nsingle*: \neg *is-singleton-list* ($\text{map } \mathcal{T}(C, \mathcal{V}) \ ts$) **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) \vee *wf-rx* (xr, b)
by (*auto split: if-splits*)
with xts **have** *wf-ts2* $ts \vee$ *wf-ts* ts **unfolding** *wf-rx3-def* *wf-rx2-def*
wf-rx-def
by *auto*
hence $ts \neq []$ **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  $t'$ :  $t' \in \text{set } ts'$  and  $\text{diff}$ :  $\mathcal{T}(C, \mathcal{V}) \ t \neq \mathcal{T}(C, \mathcal{V}) \ t'$  by
force
from  $\text{split-list}[OF \ t']$  obtain  $\text{bef}$   $\text{aft}$  where  $ts'$ :  $ts' = \text{bef} \ @ \ t' \ \# \ \text{aft}$ 
by auto
from  $\text{split-list}[OF \ xts]$  obtain  $\text{bef}'$   $\text{aft}'$  where  $xr$ :  $xr = \text{bef}' \ @ \ (x, \ ts)$ 
 $\# \ \text{aft}'$  by auto
obtain  $M'$  where  $mp$ :  $mp\text{-lr } mp = \text{add-mset} \ (t, \ \text{Var } x) \ (\text{add-mset} \ (t', \ \text{Var } x) \ M')$ 
unfolding  $mp \ ts \ ts' \ xr$ 
unfolding  $mp\text{-lr-def}$  by (auto simp: mp-rx-def)
show  $\text{match-fail} \ (mp\text{-lr } mp)$  unfolding  $mp$ 
by (rule match-clash-sort[OF diff])
qed
also have  $\{\# \ M + \text{pat-lr } mps \ \#\} + P = \text{add-mset} \ (M + \text{pat-lr } mps)$ 
P by auto
also have  $(\dots, \text{add-mset} \ (M + \text{pat-lr} \ (\text{filter } ?f \ mps)) \ P) \in \Rightarrow^*$ 
by (rule Cons(1), insert Cons, auto)
also have  $M + \text{pat-lr} \ (\text{filter } ?f \ mps) = M + \text{pat-lr} \ (\text{filter } ?f \ (mp \ \# \ mps))$ 
using False by auto
finally show  $?thesis$  .
qed
qed auto
from  $\text{steps}[\text{unfolded } p'\text{-split}] \ \text{steps2}$ 
have  $\text{steps}: (\text{add-mset} \ (\text{pat-mset-list } p) \ P, \ \text{add-mset} \ (M + \text{pat-lr } P') \ P) \in$ 
 $\Rightarrow^*$  by auto
have  $\text{step}: (\text{add-mset} \ (M + \text{pat-lr } P') \ P, \ \{\# \ \text{pat-lr } P' \ \#\} + P) \in \Rightarrow^=$ 
proof (cases ivc = [])
case True
thus  $?thesis$  unfolding  $M\text{-def } \text{pat-lr-def}$  by auto
next
case  $ivc\text{-ne}: \text{False}$ 
have  $(\text{add-mset} \ (M + \text{pat-lr } P') \ P, \ \{\# \ \text{pat-lr } P' \ \#\} + P) \in \Rightarrow$ 
unfolding  $P\text{-step-def}$ 
proof (standard, unfold split, rule P-simp-pp, rule pat-inf-var-conflict)
from  $ivc\text{-ne}$ 
obtain  $lx \ rx \ b \ ivc'$  where  $ivc$ :  $ivc = (lx, rx, b) \ \# \ ivc'$  by (cases ivc, auto)
hence  $(lx, rx, b) \in \text{set } ivc$  by auto
from  $ivc\text{-b}[OF \ \text{this}]$  have  $mp\text{-rx} \ (rx, b) \neq \{\#\}$  unfolding  $\text{inf-var-conflict-def}$ 
by auto
thus  $M \neq \{\#\}$  unfolding  $M\text{-def } ivc \ \text{pat-lr-def}$  by auto
next
{
fix  $xl \ xr \ b$ 
assume  $(xl, xr, b) \in \text{set } ivc$ 
from  $ivc\text{-b}[OF \ \text{this}]$  have  $\text{inf-var-conflict} \ (mp\text{-mset} \ (mp\text{-rx} \ ((xr, b))))$ 
by simp
hence  $\text{inf-var-conflict} \ (mp\text{-mset} \ (mp\text{-lr} \ (xl, xr, b)))$ 
unfolding  $mp\text{-lr-def } \text{inf-var-conflict-def}$  by force

```

```

    }
    thus Ball (pat-mset M) inf-var-conflict unfolding M-def pat-lr-def by
auto
next
  show  $\forall x \in \text{tvars-pat } (\text{pat-mset } (\text{pat-lr } P')). \neg \text{inf-sort } (\text{snd } x)$  by fact
qed (insert impr, auto)
thus ?thesis ..
qed
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)  $\in \Rightarrow^*$ 
  using steps step unfolding pat-mset-list-lr by auto
note res = res[folded P'-def]
show ?thesis
proof (intro conjI impI)
  assume res = Fin-Constr-Form fcf
  with res have fcf: fcf = map (map snd  $\circ$  fst  $\circ$  snd) P' by auto
  have lr-rx: map mp-lr-list P' = map (mp-rx-list  $\circ$  snd) P'
  proof (intro map-cong[OF refl])
    show mp  $\in$  set P'  $\implies$  mp-lr-list mp = (mp-rx-list  $\circ$  snd) mp for mp
    unfolding mp-lr-list-def using all-lhs-Var unfolding P'-def by (cases
mp, auto)
  qed
  have tvars: tvars-pat (pat-mset (pat-mset-list (map mp-lr-list P'))) =
tvars-spat (set3 fcf)
  (is - = ?VV)
  unfolding fcf lr-rx tvars-pat-def tvars-match-def mp-rx-list-def pat-mset-list
pat-list-def
  by force
  have VV: ?VV  $\subseteq$  tvars-pat (pat-mset (pat-lr p')) unfolding tvars[symmetric]
pat-mset-list pat-mset-list-lr
  by (rule tvars-pat-mono, insert P', auto simp: pat-lr-def)
  also have ...  $\subseteq$  {.. $n$ }  $\times$  S by fact
  finally have VSS: ?VV  $\subseteq$  SS ?VV  $\subseteq$  {.. $n$ }  $\times$  S by force+

  have pat-complete C (pat-mset (pat-mset-list (map mp-lr-list P')))
 $\longleftrightarrow$  simple-pat-complete C ?VV (set3 fcf)
  unfolding pat-complete-def simple-pat-complete-def tvars fcf
proof (intro all-cong, unfold pat-mset-list lr-rx, goal-cases)
  case (1  $\sigma$ )
  show Bex (pat-list (map (mp-rx-list  $\circ$  snd) P')) (match-complete-wrt  $\sigma$ )
=
  Bex (set3 (map (map snd  $\circ$  fst  $\circ$  snd) P')) (simple-match-complete-wrt
 $\sigma$ ) (is ?l = ?r)
proof
  assume ?l
  from this[unfolded pat-list-def] obtain mp

```

where $mp: mp \in \text{set } P'$ **and** $comp: \text{match-complete-wrt } \sigma$ ($\text{set } (mp\text{-rx-list } (snd\ mp))$) **by** *auto*
obtain $lx\ rx\ b$ **where** $mp\text{-id}: mp = (lx,rx,b)$ **by** (*cases mp, auto*)
from mp **have** $set2: \text{set2 } ((\text{map } snd \circ \text{fst} \circ \text{snd})\ mp) \in \text{set3 } (\text{map } (\text{map } snd \circ \text{fst} \circ \text{snd})\ P')$
unfolding *o-def image-comp set-map* **by** *auto*
from $comp$ [*unfolded match-complete-wrt-def*] **obtain** μ **where**
 $comp: (t, l) \in \text{set } (mp\text{-rx-list } (rx,b)) \implies t \cdot \sigma = l \cdot \mu$ **for** $t\ l$
by (*auto simp: mp-id*)
show $?r$
proof (*intro beXI[OF - set2], unfold mp-id o-def snd-conv fst-conv*
set-map)
show $\text{simple-match-complete-wrt } \sigma$ ($\text{set } 'snd\ 'set\ rx$)
unfolding *simple-match-complete-wrt-def*
proof (*intro ballI impI allI UNIQ-subst-pairI*)
fix $eqc\ s\ t$
assume $eqc: eqc \in \text{set } 'snd\ 'set\ rx$ **and** $st: s \in eqc\ t \in eqc$
from eqc **obtain** $x\ ts$ **where** $xts: (x,ts) \in \text{set } rx$ **and** $eqc: eqc = \text{set } ts$
ts by auto
from $xts\ st$ **have** $\{(s, \text{Var } x), (t, \text{Var } x)\} \subseteq \text{set } (mp\text{-rx-list } (rx,b))$
unfolding *mp-rx-list-def eqc* **by** *auto*
with $comp$ **show** $s \cdot \sigma = t \cdot \sigma$ **by** *auto*
qed
qed
next
assume $?r$
from $this$ [*simplified*] **obtain** mp
where $mp: mp \in \text{set } P'$
and $comp: \text{simple-match-complete-wrt } \sigma$ ($\text{set } 'snd\ 'set\ (\text{fst } (snd\ mp))$) **by** *auto*
obtain $b\ lx\ rx$ **where** $mp\text{-id}: mp = (lx,rx,b)$ **by** (*cases mp, auto*)
have $comp: \text{simple-match-complete-wrt } \sigma$ ($\text{set } 'snd\ 'set\ rx$) **using**
mp-id comp by auto
from mp **have** $mem: (\text{set} \circ (mp\text{-rx-list} \circ \text{snd}))\ mp \in \text{pat-list } (\text{map } (mp\text{-rx-list} \circ \text{snd})\ P')$
unfolding *pat-list-def* **by** *auto*
define μ **where** $\mu\ x = (\text{case } \text{map-of } rx\ x\ \text{of } \text{Some } ts \Rightarrow \text{hd } ts \cdot \sigma)$ **for** x
show $?l$
proof (*intro beXI[OF - mem], unfold mp-id o-def snd-conv*)
show $\text{match-complete-wrt } \sigma$ ($\text{set } (mp\text{-rx-list } (rx, b))$) **unfolding**
match-complete-wrt-def
proof (*intro exI[of - μ], clarify*)
fix $t\ l$
assume $(t,l) \in \text{set } (mp\text{-rx-list } (rx, b))$
from $this$ [*unfolded mp-rx-list-def*]
obtain $x\ ts$ **where** $xts: (x,ts) \in \text{set } rx$ **and** $t: t \in \text{set } ts$ **and** $l: l =$
Var x by auto
from xts **have** $\text{set } ts \in \text{set } 'snd\ 'set\ rx$ **by** *force*

```

      note comp = comp[unfolded simple-match-complete-wrt-def,
rule-format, OF this]
      note comp = UNIQ-subst-pairD[OF comp]
      from wf[unfolded wf-pat-lr-def] P' mp have wf: wf-lr3 mp by auto
      from wf[unfolded wf-lr3-def mp-id wf-rx3-def wf-rx2-def wf-rx-def]
      have dist: distinct (map fst rx) by (auto split: if-splits)
      hence  $l \cdot \mu = hd\ ts \cdot \sigma$  unfolding  $\mu$ -def using xts by (simp add: l)
      also have  $\dots = t \cdot \sigma$  using comp[of hd ts t] t by (cases ts, auto)
      finally show  $t \cdot \sigma = l \cdot \mu$  by simp
    qed
  qed
  qed
  also have  $\dots \iff$  simple-pat-complete C SS (set3 fcf) (is  $?l = ?r$ )
  proof –
    obtain VV where VV[no-atp]:  $?VV = VV$  by auto
    {
      assume  $?l$ 
      have  $?r$  unfolding simple-pat-complete-def
      proof (intro allI impI)
        fix  $\sigma$ 
        assume  $\sigma :_s \mathcal{V} \mid' SS \rightarrow \mathcal{T}(C)$ 
        with VSS have  $\sigma :_s \mathcal{V} \mid' ?VV \rightarrow \mathcal{T}(C)$ 
          by (meson restrict-map-mono-right sorted-map-cmono)
        from  $\langle ?l \rangle$ [unfolded simple-pat-complete-def, rule-format, OF this]
        show Bex (set3 fcf) (simple-match-complete-wrt  $\sigma$ ) .
      qed
    }
  }
  moreover
  {
    assume  $?r$ 
    have  $?l$  unfolding simple-pat-complete-def VV
    proof (intro allI impI)
      fix  $\sigma$ 
      assume  $\sigma :_s \mathcal{V} \mid' VV \rightarrow \mathcal{T}(C)$ 
      define  $\delta$  where  $\delta\ x =$  (if  $x \in VV$  then  $\sigma\ x$  else  $\sigma\ g'\ x$ ) for  $x$ 
      have  $\delta :_s \mathcal{V} \mid' SS \rightarrow \mathcal{T}(C)$ 
      proof
        fix  $x\ \iota$ 
        assume  $x : \iota$  in  $\mathcal{V} \mid' SS$ 
        then obtain  $v$  where  $xv : x = (v, \iota)$  and  $\iota : \iota \in S$ 
          by (cases x, auto simp: hastype-def restrict-map-def split: if-splits)
        from not-empty-sort[OF  $\iota$ ]
        have  $\iota : \exists\ t. t : \iota$  in  $\mathcal{T}(C)$  unfolding empty-sort-def by auto
        show  $\delta\ x : \iota$  in  $\mathcal{T}(C)$ 
        proof (cases x  $\in VV$ )
          case True
            hence  $\delta\ x = \sigma\ x$  by (auto simp:  $\delta$ -def)
            with  $\sigma\ True\ x$  show  $?thesis$ 
          case False
            show  $\delta\ x : \iota$  in  $\mathcal{T}(C)$  by auto
        qed
      qed
    }
  }

```

```

      by (metis hastype-restrict sorted-map-def)
    next
      case False
      hence  $\delta x = \sigma g' x$  unfolding  $xv$   $\delta$ -def by auto
      thus ?thesis using  $\sigma g' x$  by (metis sorted-map-def)
    qed
  qed
  from <?r>[unfolded simple-pat-complete-def, rule-format, OF this]
  obtain  $mp$  where  $mp: mp \in set3 fcf$  and  $comp: simple-match-complete-wrt$ 
 $\delta mp$  by auto
  have  $simple-match-complete-wrt \delta mp = simple-match-complete-wrt$ 
 $\sigma mp$ 
  unfolding simple-match-complete-wrt-def UNIQ-subst-def
  proof (intro ball-cong refl all-cong1 imp-cong arg-cong[of - - UNIQ]
image-cong)
    fix  $eqc t$ 
    assume  $eqc \in mp t \in eqc$ 
    hence  $vars: vars t \subseteq VV$  unfolding  $VV[symmetric]$  using  $mp$  by
force
    show  $t \cdot \delta = t \cdot \sigma$ 
      by (rule term-subst-eq, insert vars, auto simp:  $\delta$ -def)
    qed
    thus  $Bex (set3 fcf) (simple-match-complete-wrt \sigma)$  using  $comp mp$ 
by auto
  qed
}
ultimately show ?thesis by blast
qed
finally have  $equiv: pat-complete C (pat-mset (pat-mset-list (map mp-lr-list$ 
 $P')) = simple-pat-complete C SS (set3 fcf) .$ 

  have  $length fcf \leq length P'$ 
  unfolding  $fcf$  by simp
  also have  $\dots \leq length no-ivc$ 
  unfolding  $P'$ -def by simp
  also have  $\dots \leq length p'$ 
  unfolding  $Notf$  by simp
  also have  $\dots < k$ 
  using  $lvar-p'$  unfolding  $size-cond-pp-def pat-lr-def$  by simp
  finally have  $lenk: length fcf < k .$ 

{
  fix  $smp$ 
  assume  $smp \in set3 fcf$ 
  from  $this[unfolded fcf]$  obtain  $mp$  where  $mpP': mp \in set P'$ 
  and  $smp: smp = set2 ((map snd o fst o snd) mp)$  by auto
  obtain  $lx rx b$  where  $mp-id: mp = (lx,rx,b)$  by (cases  $mp$ , auto)
  with  $smp$  have  $smp: smp = set2 (map snd rx)$  by auto
  from  $wf[unfolded wf-pat-lr-def] P' mpP'$  have  $wf: wf-lr3 mp$  by auto

```

```

from mpP'[unfolded P'-def] have mp: mp ∈ set no-ivc and fmp: ?f mp
by auto
from no-ivc-b[OF mp] all-lhs-Var mp mp-id
have 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 is-singleton-list2]
from wf[unfolded mp-id wf-lr3-def split] have wf-rx3 (rx, False) by auto
from this[unfolded wf-rx3-def] impr have wf: ∀ xts ∈ set rx. wf-ts3 (snd
xts) by auto
have finite-constructor-form-mp smp unfolding smp finite-constructor-form-defs
proof (intro ballI)
fix eqc
assume eqc ∈ set2 (map snd rx)
then obtain x ts where xts: (x,ts) ∈ set rx and eqc: eqc = set ts by
auto
from wf[rule-format, OF xts] have wf-ts3 ts by auto
from this[unfolded wf-ts3-def] eqc obtain y where y: Var y ∈ set ts
by auto
from fmp[rule-format, OF xts] y have T(C,ℳ) ' eqc = {T(C,ℳ) (Var
y)}
unfolding snd-conv set-map eqc[symmetric]
by (metis empty-iff insert-iff insert-image)
also have T(C,ℳ) (Var y) = Some (snd y) by auto
finally have eq: T(C,ℳ) ' eqc = {Some (snd y)} .
show eqc ≠ {} ∧ (∃ι. finite-sort C ι ∧ (∀ t∈eqc. t : ι in T(C,ℳ |' SS)))

proof (intro exI[of - snd y] conjI)
show eqc ≠ {} unfolding eqc using y by auto
show ∀ t∈eqc. t : snd y in T(C,ℳ |' SS)
proof
fix t
assume t: t ∈ eqc
with eq have tCV: t : snd y in T(C,ℳ) unfolding hastype-def by
blast
from t[unfolded eqc] xts mpP'[unfolded mp-id]
have vars t ⊆ ?VV unfolding fcf by force
with VSS have vars t ⊆ SS by auto
with tCV show t : snd y in T(C,ℳ |' SS)
by (meson hastype-in-Term-mono-right hastype-in-Term-restrict-vars
restrict-map-mono-right)
qed
have y: y ∈ tvars-pat (pat-mset (pat-lr P'))
unfolding tvars-pat-def tvars-match-def
pat-lr-def mp-lr-def mp-rx-def
apply clarsimp
apply (intro bexI[OF - mpP'[unfolded mp-id]])
apply (unfold split, clarsimp)
apply (intro bexI[OF - xts])

```

```

      apply (simp)
      by (rule exI[of - Var y], insert y, auto)
    from no-inf-sort[rule-format, OF this]
    have ninf:  $\neg$  inf-sort (snd y) by auto
    have  $y \in$  tvars-pat (pat-mset (pat-lr p'))
      using P' y unfolding tvars-pat-def pat-lr-def by force
    with varsp' have snd y  $\in$  S by auto
    from inf-sort[OF this] ninf
    show finite-sort C (snd y) by simp
  qed
}
with steps equiv VSS(2) lenk show  $\exists P'. (add-mset (pat-mset-list p) P,$ 
 $add-mset P' P) \in \Rightarrow^*$ 
   $\wedge$  finite-constructor-form-pat (set3 fcf)
   $\wedge$  tvars-spat (set3 fcf)  $\subseteq$   $\{..<n\} \times S$ 
   $\wedge$  length fcf  $< k$ 
   $\wedge$  pat-complete C (pat-mset P') = simple-pat-complete C SS (set3 fcf)
  unfolding finite-constructor-form-pat-def
  by blast
qed (insert res impr, auto)
qed
qed
qed
qed

```

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

lemma *pats-impl*: **assumes** $\forall p \in$ set ps. tvars-pat (pat-list p) \subseteq $\{..<n\} \times S$
and $\forall pp \in$ set ps. lvar-cond-pp nl (pat-mset-list pp) \wedge size-cond-pp (pat-mset-list pp)
and $\forall pp \in$ 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 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 renNat fcf-solve CC n nl ps, unfolded Cons
list.simps, folded Cons]
from 1(2)[rule-format, of p] Cons have tvars-pat (pat-list p)  $\subseteq \{..<n\} \times S$ 
by auto
note pat-impl = pat-impl[OF refl this]
from 1(3) have  $\forall pp \in \# \text{ add-mset } (pat\text{-mset-list } p) (pats\text{-mset-list } ps1).$ 
lvar-cond-pp nl pp \wedge size-cond-pp pp
unfolding Cons by auto
note pat-impl = pat-impl[OF this, folded id]
let ?step = ( $\Rightarrow$ ) :: ((f, 'v, 's)pats-problem-mset  $\times$  (f, 'v, 's)pats-problem-mset)set

{
from rel-P-trans have single: ?step  $\subseteq (\prec\text{mul})^{\wedge-1}$ 
unfolding P-step-def by auto
have  $(s,t) \in ?step^{\wedge+} \implies (t,s) \in (\prec\text{mul})^{\wedge+}$   $(s,t) \in ?step^{\wedge*} \implies (t,s) \in$ 
 $(\prec\text{mul})^{\wedge*}$  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]
have steps: (pats-mset-list ps, add-mset {#} (pats-mset-list ps1))  $\in \Rightarrow^*$ 
by auto
show ?thesis
proof (cases add-mset {#} (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 False]
have (add-mset {#} (pats-mset-list ps1), bottom-mset)  $\in \Rightarrow$  unfolding
P-step-def by auto
with steps have (pats-mset-list ps, bottom-mset)  $\in \Rightarrow^*$  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))*) $\in \Rightarrow^+$
and *vars*: *tvars-pat* (\bigcup (*pat-list ' set ps2*)) $\subseteq \{..<n2\} \times S$
and *lvars*: ($\forall pp \in \#mset$ (*map pat-mset-list ps2*) + *pats-mset-list ps1*).
lvar-cond-pp nl2 pp \wedge *size-cond-pp pp*)
and *n2*: $n \leq n2$ **by** *auto*
from *steps-to-rel*(1)[*OF steps*] **have** *rel*: (*ps2 @ ps1*, *ps*) \in *inv-image* (\prec_{mul^+})
pats-mset-list
by *auto*
have *vars*: $\forall p \in set$ (*ps2 @ ps1*). *tvars-pat* (*pat-list p*) $\subseteq \{..<n2\} \times S$
proof
fix *p*
assume $p \in set$ (*ps2 @ ps1*)
hence $p \in set$ *ps2* \vee $p \in set$ *ps1* **by** *auto*
thus *tvars-pat* (*pat-list p*) $\subseteq \{..<n2\} \times S$
proof
assume $p \in set$ *ps2*
hence *tvars-pat* (*pat-list p*) \subseteq *tvars-pat* (\bigcup (*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*) \wedge
size-cond-pp (*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** $\dots =$ *pats-complete C* (*pat-list ' set (ps2 @ ps1)*)
using *mp*[*OF IH*][*OF rel vars lvars*] *wf2* .
also **have** $\dots =$ *pats-complete C* (*pat-list ' set ps*) **using** *steps-equiv*
unfolding *pats-mset-list*[*symmetric*] **by** *auto*
finally **show** *?thesis* .
next
case *FCF*: (*Fin-Constr-Form fuf*)
note *res* = *res*[*unfolded FCF pat-impl-result.simps*]
from *pat-impl*(3)[*OF FCF*]
obtain *p'* **where** *steps*: (*pats-mset-list ps*, *add-mset p' (pats-mset-list ps1)*) \in
 \Rightarrow^*

```

and fcf: finite-constructor-form-pat (set3 fvf)
and vars: tvars-spat (set3 fvf)  $\subseteq \{..<n\} \times S$ 
and p': pat-complete C (pat-mset p') = simple-pat-complete C SS (set3 fvf)
and len: length fvf < k
and impr: improved
by auto
from vars have vars: tvars-spat (set3 fvf)  $\subseteq \{..<n\} \times UNIV$  by blast
let ?solver = fcf-solve
from fcf-solve[unfolded fcf-solver-def, rule-format, OF impr fcf vars len] p'
have p': pat-complete C (pat-mset p') = ?solver n fvf 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 (pats-mset-list ps1))
  and red: pats-complete C (pats-mset (pats-mset-list ps)) =
    pats-complete C (pats-mset (add-mset p' (pats-mset-list ps1)))
  and wf-pat (pat-mset p') unfolding wf-pats-def by auto
have red: pats-complete C (pats-mset (pats-mset-list ps))
  = (?solver n fvf  $\wedge$  pats-complete C (pats-mset (pats-mset-list ps1)))
  unfolding red p'[symmetric] by auto
have (pats-mset-list ps1, add-mset p' (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  $\forall p \in \text{set } ps1. \text{tvars-pat } (\text{pat-list } p) \subseteq \{..<n\} \times S$  by
auto
note IH = IH[OF this]
from 1(3) Cons have  $\forall pp \in \text{set } ps1. \text{lvar-cond-pp } nl (\text{pat-mset-list } pp) \wedge$ 
size-cond-pp (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
show ?thesis unfolding res red[unfolded pats-mset-list] IH[symmetric] by
auto
qed
qed
qed

```

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

corollary *pat-complete-impl*:

```

assumes wf: snd ‘  $\bigcup (\text{vars } \text{'fst } \text{'set } (\text{concat } (\text{concat } P))) \subseteq S$ 
and k: k = compute-k-parameter P
shows Pat-complete-impl (P :: (f, 'v, 's)pats-problem-list)  $\longleftrightarrow$  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:  $\forall p \in \text{set } P. \text{tvars-pat } (\text{pat-list } p) \subseteq \{..<n\} \times S$ 
proof (safe)
  fix p x  $\iota$ 
  assume p:  $p \in \text{set } P$  and xp:  $(x, \iota) \in \text{tvars-pat } (\text{pat-list } p)$ 
  hence  $x \in \text{set } ?l$  unfolding 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  $\iota \in S$  by (auto simp: wf-pat-iff)
qed
show ?thesis
proof (cases improved)
  case False
  have 0:  $\forall p \in \text{set } P. \text{lvar-cond-pp } 0 (\text{pat-mset-list } p) \wedge \text{size-cond-pp } (\text{pat-mset-list } p)$ 
  unfolding lvar-cond-pp-def lvar-cond-def allowed-vars-def size-cond-pp-def
  using False k compute-k-parameter by (auto simp: pat-mset-list-def)
  have Pat-complete-impl P = Pats-impl n 0 P
  unfolding pat-complete-impl-def n-def Let-def using False k 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 k using True by auto
  also have ... = pats-complete C (pat-list ' set ?Q)
  proof (rule pats-impl)
  show  $\forall p \in \text{set } ?Q. \text{tvars-pat } (\text{pat-list } p) \subseteq \{..<n\} \times S$ 
  proof
    fix rp
    assume rp  $\in \text{set } ?Q$ 
    then obtain p where p:  $p \in \text{set } P$  and rp:  $rp = ?r p$  by auto
    have id:  $\text{tvars-pat } (\text{pat-list } rp) = \text{tvars-pat } (\text{pat-list } p)$ 
    unfolding pat-list-def rp tvars-pat-def tvars-match-def by force
    with n p show  $\text{tvars-pat } (\text{pat-list } rp) \subseteq \{..<n\} \times S$  by auto
  qed
  show Ball (pat-list ' set ?Q) wf-pat
  proof -
  {
    fix rp
    assume rp:  $rp \in \text{set } ?Q$ 

```

```

    then obtain  $p$  where  $p \in \text{set } P$  and  $rp: rp = ?r p$  by auto
    from  $\text{this}(1)$  wf have  $\text{wf-pat } (\text{pat-list } p)$  by auto
    hence  $\text{wf-pat } (\text{pat-list } rp)$  unfolding  $\text{wf-pat-def wf-match-def}$ 
      unfolding  $\text{pat-list-def } rp$   $\text{tvars-match-def}$  by force
  }
  thus ?thesis by blast
qed
show  $\forall p \in \text{set } ?Q. \text{lvar-cond-pp } 0 (\text{pat-mset-list } p) \wedge \text{size-cond-pp } (\text{pat-mset-list } p)$ 
)
proof
  fix  $rp$ 
  assume  $\text{mem}: rp \in \text{set } ?Q$ 
  then obtain  $p$  where  $rp: rp = ?r p$  by auto
  show  $\text{lvar-cond-pp } 0 (\text{pat-mset-list } rp) \wedge \text{size-cond-pp } (\text{pat-mset-list } rp)$ 
    unfolding  $\text{lvar-cond-pp-def lvar-cond-def}$ 
  proof (intro conjI subsetI)
    from  $k$  mem
  show  $\text{size-cond-pp } (\text{pat-mset-list } rp)$  unfolding  $\text{size-cond-pp-def pat-mset-list-def}$ 
    using  $\text{compute-k-parameter[of } P]$  by auto
  fix  $x$ 
  assume  $x \in \text{lvars-pp } (\text{pat-mset-list } rp)$ 
  from  $\text{this}[\text{unfolded lvars-pp-def lvars-mp-def pat-mset-list-def } rp]$ 
  obtain  $t :: ('f, 'v)$  term where  $x \in \text{vars } (\text{map-vars renVar } t)$  by auto
  hence  $x \in \text{range renVar}$  by (induct  $t$ , auto)
  thus  $x \in \text{allowed-vars } 0$  unfolding  $\text{allowed-vars-def}$  by auto
qed
qed
qed
also have  $\dots = \text{pats-complete } C (\text{pat-list } ' \text{set } P)$ 
  unfolding  $\text{set-map image-comp}$ 
  unfolding  $\text{Ball-image-comp o-def}$ 
proof (intro ball-cong[OF refl])
  fix  $p$ 
  assume  $p: p \in \text{set } P$ 
  have  $\text{id}: \text{pat-list } (\text{map } (\text{map } (\text{apsnd } (\text{map-vars renVar}))) p) = (\lambda mp. \text{apsnd } (\text{map-vars renVar}) ' mp) ' \text{pat-list } p$ 
    unfolding  $\text{pat-list-def set-map image-comp o-def ..}$ 
  note  $\text{bex} = \text{bex-simps}(7)$ 
  from  $\text{wf } p$ 
  have  $\text{wfp}: \text{wf-pat } (\text{pat-list } p)$ 
    by (fastforce simp: subset-iff wf-pat-def wf-match-def tvars-match-def)
  from  $\text{wf } p$ 
  have  $\text{wf2}: \text{wf-pat } (\text{pat-list } (?r p))$ 
    by (fastforce simp: id subset-iff wf-pat-def wf-match-def tvars-match-def)
  show  $\text{pat-complete } C (\text{pat-list } (?r p)) = \text{pat-complete } C (\text{pat-list } p)$ 
    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  $\sigma mp$ 

```

```

have id: map-vars renVar = ( $\lambda t. t \cdot (\text{Var } o \text{ renVar})$ ) using map-vars-term-eq[of renVar] by auto
show match-complete-wrt  $\sigma$  (apsnd (map-vars renVar) ' mp) = match-complete-wrt
 $\sigma$  mp (is ?m1 = ?m2)
proof
  assume ?m1
  from this[unfolded match-complete-wrt-def] obtain  $\mu$ 
  where match:  $\bigwedge t l. (t, l) \in mp \implies t \cdot \sigma = \text{map-vars } renVar \ l \cdot \mu$  by
force
show ?m2 unfolding match-complete-wrt-def
  by (intro exI[of - (Var o renVar)  $\circ_s \mu$ ], insert match[unfolded id], auto)
next
  assume ?m2
  from this[unfolded match-complete-wrt-def] obtain  $\mu$ 
  where match:  $\bigwedge t l. (t, l) \in mp \implies t \cdot \sigma = l \cdot \mu$  by force
  {
    fix t
    have  $t \cdot (\text{Var } o \text{ renVar}) \cdot (\text{Var } o \text{ the-inv } renVar) \cdot \mu = t \cdot \mu$ 
    unfolding subst-subst
    proof (intro term-subst-eq)
    fix x
    from renaming-ass[rule-format, OF <improved>, unfolded renaming-funs-def]
    have inj: inj renVar by auto
    from the-inv-f-f[OF this]
    show  $((\text{Var } o \text{ renVar}) \circ_s (\text{Var } o \text{ the-inv } renVar) \circ_s \mu) \ x = \mu \ x$ 

    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)  $\circ_s \mu$ ], insert match, auto simp:
id)
  qed
qed
qed
finally show ?thesis .
qed
qed
end
end

```

7.3 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*: $distinct (\text{map } fst \ l) \implies x : \sigma \text{ in } \text{map-of } l \longleftrightarrow (x, \sigma) \in \text{set } l$

by (auto simp: hastype-def)

lemma *fun-hastype-in-map-of*: $\text{distinct } (\text{map fst } l) \implies$
 $x : \sigma s \rightarrow \tau \text{ in } \text{map-of } l \longleftrightarrow ((x, \sigma s), \tau) \in \text{set } l$
by (auto simp: fun-hastype-def)

definition *constr-list* **where** $\text{constr-list } Cs \ s = \text{map fst } (\text{filter } ((=) \ s \ o \ \text{snd}) \ Cs)$

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

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

lemma *sorts-of-ssig-list*:
assumes $((f, \sigma s), \tau) \in \text{set } Cs$
shows $\text{set } \sigma s \subseteq \text{set } (\text{sorts-of-ssig-list } Cs) \ \tau \in \text{set } (\text{sorts-of-ssig-list } Cs)$
using *assms*
by (auto simp: sorts-of-ssig-list-def)

definition *max-arity-list* **where**
 $\text{max-arity-list } Cs = \text{max-list } (\text{map } (\text{length } o \ \text{snd } o \ \text{fst}) \ Cs)$

lemma *max-arity-list*:
 $((f, \sigma s), \tau) \in \text{set } Cs \implies \text{length } \sigma s \leq \text{max-arity-list } Cs$
by (force simp: max-arity-list-def o-def intro! :max-list)

locale *pattern-completeness-list* =
fixes Cs **and** $k :: \text{nat}$
assumes *dist*: $\text{distinct } (\text{map fst } Cs)$
and *inhabited*: $\text{decide-nonempty-sorts } (\text{sorts-of-ssig-list } Cs) \ Cs = \text{None}$
and *k1*: $k > 1$
begin

lemma *nonempty-sort*: $\bigwedge \sigma. \sigma \in \text{set } (\text{sorts-of-ssig-list } Cs) \implies \neg \text{empty-sort } (\text{map-of } Cs) \ \sigma$
using *decide-nonempty-sorts(1)[OF dist inhabited]*
by (auto elim: not-empty-sortE)

lemma *compute-inf-sorts*: $\sigma \in \text{compute-inf-sorts } Cs \longleftrightarrow \neg \text{finite-sort } (\text{map-of } Cs) \ \sigma$
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*: $\text{snd } (\text{compute-inf-card-sorts-bnd } k \ Cs) = \text{min } k \ o \ \text{card-of-sort } (\text{map-of } Cs)$
apply (*rule compute-inf-card-sorts-bnd(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-bnd k 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) ∈ set (constr-list Cs s)
    hence ((f,ss),s) ∈ set Cs unfolding constr-list-def by auto
    from max-arity-list[OF this] have length ss ≤ max-arity-list Cs by auto
  }
  then show m: ∀ a ∈ length ‘ snd ‘ set (constr-list Cs s). a ≤ max-arity-list Cs for
s by auto
  show k > 1 by (rule k1)
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 ⇒ ('f × 's list)list — a function to compute all constructors of
given sort as list
  and Is :: 's ⇒ bool — a function to indicate whether a sort is infinite
  and Cd :: 's ⇒ nat — a function to compute finite cardinality of sort
begin

```

definition pat-complete-impl-fscd = pattern-completeness-context.pat-complete-impl
m Cl Is False undefined undefined undefined undefined

definition pats-impl-fscd = pattern-completeness-context.pats-impl m Cl Is False

undefined undefined undefined

definition *pat-impl-fscd* = *pattern-completeness-context.pat-impl m Cl Is False undefined undefined*

definition *pat-inner-impl-fscd* = *pattern-completeness-context.pat-inner-impl Is False undefined*

definition *match-decomp'-impl-fscd* = *pattern-completeness-context.match-decomp'-impl Is False undefined*

definition *find-var-fscd* :: (*f,v,s*)*match-problem-lr list* ⇒ - **where**

find-var-fscd 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-fscd*: *find-var False p* = *find-var-fscd p*

unfolding *find-var-fscd-def find-var-def if-False* **by** (*auto split: list.splits*)

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

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

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

lemma *pats-impl-fscd-code*[*code*]:

pats-impl-fscd n nl ps =
 (case *ps* of [] ⇒ *True*
 | *p # ps1* ⇒
 (case *pat-impl-fscd n nl p* of *Incomplete* ⇒ *False*
 | *New-Problems (n', nl', ps2)* ⇒ *pats-impl-fscd n' nl' (ps2 @ ps1)*))

unfolding *pats-impl-fscd-def pattern-completeness-context.pats-impl.simps*[of - -
- - - - - *ps*]

unfolding *pat-impl-fscd-def*[*symmetric*]

unfolding *pat-impl-fscd-code*

by (*auto split: list.splits option.splits*)

lemmas *match-decomp'-impl-fscd-code*[*code*] =

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

lemmas *pat-inner-impl-fscd-code*[*code*] =

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

```

context
  fixes
     $C :: ('f \times 's \text{ list}) \Rightarrow 's \text{ option}$ 
    and  $rn :: nat \Rightarrow 'v$ 
    and  $rv :: 'v \Rightarrow 'v$ 
    and  $fcf\text{-solve} :: nat \Rightarrow ('f, 's)\text{spp-list} \Rightarrow bool$ 
  begin
    definition  $pat\text{-complete-impl-fcf} = pattern\text{-completeness-context.pat-complete-impl}$ 
     $m \ Cl \ Is \ True \ rn \ rv \ fcf\text{-solve} \ C$ 
    definition  $pats\text{-impl-new} = pattern\text{-completeness-context.pats-impl} \ m \ Cl \ Is \ True$ 
     $rn \ fcf\text{-solve} \ C$ 
    definition  $pat\text{-impl-new} = pattern\text{-completeness-context.pat-impl} \ m \ Cl \ Is \ True \ rn$ 
     $C$ 
    definition  $pat\text{-inner-impl-new} = pattern\text{-completeness-context.pat-inner-impl} \ Is \ True$ 
     $rn$ 
    definition  $match\text{-decomp}'\text{-impl-new} = pattern\text{-completeness-context.match-decomp}'\text{-impl}$ 
     $Is \ True \ rn$ 
    definition  $find\text{-var-new} = find\text{-var} \ True$ 

    lemmas  $pat\text{-complete-impl-fcf-code}[code] = pattern\text{-completeness-context.pat-complete-impl-def}[of$ 
     $m \ Cl \ Is \ True \ rn \ rv \ fcf\text{-solve} \ C,$ 
     $\text{folded } pat\text{-complete-impl-fcf-def} \ pats\text{-impl-new-def},$ 
     $\text{unfolded } if\text{-True} \ Let\text{-def}]$ 

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

    lemmas  $pats\text{-impl-new-code}[code] = pattern\text{-completeness-context.pats-impl.simps}[of$ 
     $m \ Cl \ Is \ True \ rn \ fcf\text{-solve} \ C,$ 
     $\text{folded } pats\text{-impl-new-def} \ pat\text{-impl-new-def}]$ 

    lemma  $match\text{-decomp}'\text{-impl-new-code} [code]:$ 
     $\langle match\text{-decomp}'\text{-impl-new} \ n \ mp =$ 
     $\text{map-option } (\lambda(xs, mp). \text{if case } mp \text{ of } (xl, rx, b) \Rightarrow \neg b \wedge xl = [] \text{ then } pat\text{-}$ 
     $tern\text{-completeness-context.decomp}'\text{-impl} \ rn \ n \ xs \ mp \text{ else } (n, mp))$ 
     $(pattern\text{-completeness-context.match-steps-impl} \ Is \ mp) \rangle$ 
    by ( $\text{simp add: } match\text{-decomp}'\text{-impl-new-def} \ pattern\text{-completeness-context.match-decomp}'\text{-impl-def}$ 
     $pattern\text{-completeness-context.apply-decompose}'\text{-def}$ )

    lemma  $find\text{-var-new-code}[code]:$ 
     $\langle find\text{-var-new} \ p = \text{fst} \ (hd \ (List.maps \ (\lambda(lx, rx). \ lx) \ p)) \rangle$ 
    by ( $\text{simp add: } find\text{-var-new-def} \ find\text{-var-def}$ )

    lemma  $pat\text{-inner-impl-new-code} [code]:$ 
     $\langle pat\text{-inner-impl-new} \ n \ [] \ pd = \text{Some} \ (n, pd) \rangle$ 
     $\langle pat\text{-inner-impl-new} \ n \ (mp \ \# \ p) \ pd =$ 

```

(*case match-decomp'-impl-new n mp of None \Rightarrow pat-inner-impl-new n p pd*
| *Some (n', mp') \Rightarrow if empty-lr mp' then None else pat-inner-impl-new n' p*
(*mp' # pd*))
by (*auto simp add: pat-inner-impl-new-def match-decomp'-impl-new-def pattern-completeness-context.pat-inn*
split: option.split)

end

end

definition *decide-pat-complete-fscd* :: ((*f* \times *s list*) \times *s*)*list* \Rightarrow (*f, 'v, 's*)*pats-problem-list*
 \Rightarrow *bool* **where**

decide-pat-complete-fscd Cs P = (*let*
m = *max-arity-list Cs*;
Cl = *constr-list Cs*;
IS = *compute-inf-sorts Cs*
in pat-complete-impl-fscd m Cl (λ *s. s* \in *IS*)) *P*

definition *decide-pat-complete-lin* :: ((*f* \times *s list*) \times *s*)*list* \Rightarrow (*f, 'v, 's*)*pats-problem-list*
 \Rightarrow *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 ' \cup (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 2*

apply *unfold-locales*

using *dist non-empty-sorts* **by** *auto*

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-fscd*:

assumes *dist: distinct (map fst Cs)*

and *non-empty-sorts: decide-nonempty-sorts (sorts-of-ssig-list Cs) Cs = None*

and *P: snd ' \cup (vars ' fst ' set (concat (concat P))) \subseteq set (sorts-of-ssig-list*

Cs)

shows *decide-pat-complete-fscd* $Cs P = pats-complete (map-of Cs) (pat-list ' set P)$

proof –

interpret *pattern-completeness-list* $Cs compute-k-parameter P$

apply *unfold-locales*

using *dist non-empty-sorts compute-k-parameter-1* **by** *auto*

show *?thesis*

unfolding *decide-pat-complete-fscd-def Let-def pat-complete-impl-fscd-def*

apply (*rule pat-complete-impl[OF - - - P]*) **by** *auto*

qed

definition *decide-pat-complete-fcf* $:: - \Rightarrow - \Rightarrow - \Rightarrow ((f \times 's list) \times 's)list \Rightarrow (f, 'v, 's)pats-problem-list \Rightarrow bool$ **where**

decide-pat-complete-fcf $rn rv fcf-solve Cs P = (let$

$m = max-arity-list Cs;$

$Cl = constr-list Cs;$

$Cm = Mapping.of-alist Cs;$

$k = compute-k-parameter P;$

$(IS, CD) = compute-inf-card-sorts-bnd k Cs$

$in pat-complete-impl-fcf m Cl (\lambda s. s \in IS) (Mapping.lookup Cm)) rn rv$

$fcf-solve P$

definition *fvf-pp-list* $pp =$

$[[y. (t', Var y) \leftarrow pp, t' = t]. t \leftarrow remdups (map fst pp)]$

definition *fcf-list-solver* **where** *fcf-list-solver* $k Cs =$

pattern-completeness-context.fcf-solver (set (sorts-of-ssig-list Cs)) (map-of Cs)

k

theorem *decide-pat-complete-fcf*:

assumes *dist: distinct (map fst Cs)*

and *non-empty-sorts: decide-nonempty-sorts (sorts-of-ssig-list Cs) Cs = None*

and $P: snd ' \cup (vars ' fst ' set (concat (concat P))) \subseteq set (sorts-of-ssig-list Cs)$

and *ren: renaming-funs rn rv*

and *fcf-solve: fcf-list-solver (compute-k-parameter P) Cs fcf-solve*

shows *decide-pat-complete-fcf* $rn rv fcf-solve Cs P \longleftrightarrow pats-complete (map-of Cs) (pat-list ' set P)$

(**is** $?l \longleftrightarrow ?r$)

proof –

let $?k = compute-k-parameter P$

interpret *pattern-completeness-list* $Cs ?k$

apply *unfold-locales*

using *dist non-empty-sorts compute-k-parameter-1* .

have *nemp*:

$\forall f \tau s \tau \tau'. f : \tau s \rightarrow \tau$ *in* *map-of Cs* $\longrightarrow \tau' \in set \tau s \longrightarrow \neg empty-sort (map-of Cs) \tau'$

using *C-sub-S* **by** (*auto intro!*: *nonempty-sort*)

```

obtain inf cd where compute-inf-card-sorts-bnd ?k Cs = (inf,cd) by force
with compute-inf-card-sorts-bnd(2,3)[OF refl nemp dist this]
have cics: compute-inf-card-sorts-bnd ?k Cs = (compute-inf-sorts Cs, min ?k o
card-of-sort (map-of Cs))
by (simp add: Compute-Nonempty-Infinite-Sorts.compute-inf-sorts(2) dist nemp)
have Cm: Mapping.lookup (Mapping.of-alist Cs) = map-of Cs using dist
using lookup-of-alist by fastforce
have fcf: fcf-solver (compute-k-parameter P) fcf-solve
using fcf-solve[unfolded fcf-list-solver-def] .
show ?thesis
apply (unfold decide-pat-complete-fcf-def Let-def case-prod-beta cics fst-conv)
unfolding pat-complete-impl-fcf-def
by (rule pat-complete-impl[OF ren - Cm P refl], rule fcf)
qed

```

```

export-code decide-pat-complete-lin checking
export-code decide-pat-complete-fscd checking
export-code decide-pat-complete-fcf checking

```

end

```

theory FCF-Set
imports
  Pattern-Completeness-Multiset
  FCF-Problem
begin

```

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* \Rightarrow $(f, 'v, 's)$ *match-problem-set* \Rightarrow *bool* **where**

```

finite-var-form-match C mp  $\longleftrightarrow$  var-form-match mp  $\wedge$ 
 $(\forall l x y. (Var x, l) \in mp \longrightarrow (Var y, l) \in mp \longrightarrow snd x = snd y) \wedge$ 
 $(\forall l x. (Var x, l) \in mp \longrightarrow finite-sort C (snd x))$ 

```

lemma *finite-var-form-matchD*:

```

assumes finite-var-form-match C mp and  $(t, l) \in mp$ 
shows  $\exists x \iota y. t = Var (x, \iota) \wedge l = Var y \wedge finite-sort C \iota \wedge$ 
 $(\forall z. (Var z, Var y) \in mp \longrightarrow snd z = \iota)$ 
using assms by (auto simp: finite-var-form-match-def var-form-match-def)

```

definition *finite-var-form-pat* :: $(f, 's)$ *ssig* \Rightarrow $(f, 'v, 's)$ *pat-problem-set* \Rightarrow *bool* **where**
finite-var-form-pat C p = $(\forall mp \in p. finite-var-form-match C mp)$

lemma *finite-var-form-patD*:

assumes *finite-var-form-pat* C pp $mp \in pp$ $(t,l) \in mp$
shows $\exists x \iota y. t = \text{Var } (x,\iota) \wedge l = \text{Var } y \wedge \text{finite-sort } C \iota \wedge$
 $(\forall z. (\text{Var } z, \text{Var } y) \in mp \longrightarrow \text{snd } z = \iota)$
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 $pp \implies \text{var-form-pat } pp$
by (*auto simp: finite-var-form-pat-def var-form-pat-def finite-var-form-match-def*)

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

lemma *finite-var-form-pat-pat-complete*:
assumes *fvf*: *finite-var-form-pat* C pp
shows *pat-complete* C $pp \longleftrightarrow$
 $(\forall \alpha. (\forall v \in \text{tvars-pat } pp. \alpha v < \text{card-of-sort } C (\text{snd } v)) \longrightarrow$
 $(\exists mp \in pp. \forall x. \text{UNIQ } \{\alpha y \mid y. (\text{Var } y, \text{Var } x) \in mp\}))$
proof –
note $vf = \text{finite-var-form-imp-of-var-form-pat}[OF fvf]$
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 $y \iota$
assume $y: (y,\iota) \in \text{tvars-pat } pp$
from y **obtain** mp t l **where** $mp: mp \in pp$ **and** $tl:(t,l) \in mp$ **and** $yt: (y, \iota)$
 $\in \text{vars } t$
by (*auto simp: tvars-pat-def tvars-match-def*)
from *finite-var-form-patD*[*OF fvf mp tl*] yt
show *finite-sort* $C \iota$ **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

```

```

context pattern-completeness-context
begin

```

```

fun flatten-triv-sort-main :: ('f,nat × 's)term ⇒ ('f,nat × 's)term × 's where
  flatten-triv-sort-main (Var x) = (if cd-sort (snd x) = 1 then (Var (0, snd x), snd
x) else (Var x, snd x))
| flatten-triv-sort-main (Fun f ts) = (let tss = map flatten-triv-sort-main ts in
  case C (f,map snd tss) of Some s ⇒ if cd-sort s = 1 then (Var (0,s), s) else
(Fun f (map fst tss), s))

```

```

definition flatten-triv-sort :: ('f,nat × 's)term ⇒ ('f,nat × 's)term where
  flatten-triv-sort = fst o flatten-triv-sort-main

```

```

definition flatten-triv-sort-pat :: ('f,'s)simple-pat-problem ⇒ ('f,'s)simple-pat-problem
where
  flatten-triv-sort-pat = image (image (image flatten-triv-sort))

```

```

end

```

```

context pattern-completeness-context-with-assms
begin

```

```

lemma flatten-triv-sort-spp: assumes finite-constructor-form-pat p
shows finite-constructor-form-pat (flatten-triv-sort-pat p)
  simple-pat-complete C SS (flatten-triv-sort-pat p) ⟷ simple-pat-complete C
SS p

```

```

proof –

```

```

  show simple-pat-complete C SS (flatten-triv-sort-pat p) ⟷ simple-pat-complete
C SS p

```

```

  unfolding simple-pat-complete-def flatten-triv-sort-pat-def bex-simps simple-match-complete-wrt-def
ball-simps

```

```

    UNIQ-subst-def

```

```

proof (intro all-cong bex-cong refl ball-cong arg-cong[of - - UNIQ])

```

```

  fix σ mp eqc

```

```

  assume σ: σ :s V |' SS → T(C)

```

```

    and mp: mp ∈ p

```

```

    and eqc: eqc ∈ mp

```

```

  {

```

```

    fix t

```

```

assume  $t \in eqc$ 
with  $assms(1)[unfolded\ finite-constructor-form-defs, rule-format, OF\ mp\ eqc]$ 
obtain  $\iota$  where  $t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid SS)$  by auto
hence  $map-prod (\lambda t. t \cdot \sigma) id (flatten-triv-sort-main\ t) = (t \cdot \sigma, \iota)$ 
proof (induct)
  case (Var  $x\ \iota$ )
  then obtain  $v$  where  $xv: x = (v, \iota)$  and  $\iota: \iota \in S$ 
    by (auto simp: sorted-map-def restrict-map-def hastype-def split: if-splits)
  show ?case
  proof (cases cd-sort  $\iota = 1$ )
    case False
      thus ?thesis using  $xv$  by auto
    next
      case True
      from  $cd[OF\ \iota]\ True\ k1$ 
      have  $card-of-sort\ C\ \iota = 1$  by auto
      from  $this[unfolded\ card-of-sort-def]$  obtain  $s$  where  $s: \{s. s : \iota\}$  in  $\mathcal{T}(C)$ 
    by (metis card-1-singletonE)
    {
      fix  $y$ 
      have  $\sigma(y, \iota) : \iota$  in  $\mathcal{T}(C)$  using  $\sigma\ \iota$ 
      by (simp add: hastype-restrict sorted-map-def)
      with  $s$  have  $\sigma(y, \iota) = s$  by auto
    }
    thus ?thesis using True xv by simp
  qed
next
  case (Fun  $f\ ts\ \iota s\ \iota$ )
  from  $Fun(1)$  have  $C: C(f, \iota s) = Some\ \iota$  by (rule fun-hastypeD)
  from  $Fun(3)$  have  $map-snd: map\ snd (map\ flatten-triv-sort-main\ ts) = \iota s$ 
  apply (intro nth-equalityI)
  subgoal by (auto simp: list-all2-conv-all-nth)
  subgoal for  $i$  using  $arg-cong[OF\ list-all2-nthD[OF\ Fun(3),\ of\ i],\ of\ snd]$ 
by auto
  done
  show ?case
  proof (cases cd-sort  $\iota = 1$ )
    case False
      hence  $map-prod (\lambda t. t \cdot \sigma) id (flatten-triv-sort-main (Fun\ f\ ts)) =$ 
         $(Fun\ f (map (\lambda t. t \cdot \sigma) (map\ fst (map\ flatten-triv-sort-main\ ts))), \iota)$ 
      using  $map-snd\ C$  by simp
      also have  $(map (\lambda t. t \cdot \sigma) (map\ fst (map\ flatten-triv-sort-main\ ts)))$ 
         $= (map (\lambda t. t \cdot \sigma) ts)$  unfolding  $map-map\ o-def$ 
      apply (intro nth-equalityI, force)
      subgoal for  $i$  using  $arg-cong[OF\ list-all2-nthD[OF\ Fun(3),\ of\ i],\ of\ fst]$ 
    by auto
  done
  finally show ?thesis by simp

```

```

next
  case True
  hence id: map-prod ( $\lambda t. t \cdot \sigma$ ) id (flatten-triv-sort-main (Fun f ts)) = ( $\sigma$ 
( $0, \iota$ ),  $\iota$ )
    using C map-snd by simp
    from C-sub-S[OF Fun(1)] have  $\iota : \iota \in S$  by auto
    from True[unfolded cd[OF  $\iota$ ] card-of-sort-def] k1
    have card { $s. s : \iota$  in  $\mathcal{T}(C)$ } = 1 by auto
    then obtain s where  $s : \{s. s : \iota$  in  $\mathcal{T}(C)\} = \{s\}$ 
    by (metis card-1-singletonE)
    have var: Var ( $0, \iota$ ) :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \iota SS)$  using  $\iota$  by (simp add: hastype-def)

    have f: Fun f ts :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \iota SS)$  using Fun(1-2) by (metis
Fun-hastypeI)
    {
      fix t
      assume  $t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \iota SS)$ 
      hence  $t \cdot \sigma : \iota$  in  $\mathcal{T}(C)$  using  $\sigma$  by (metis subst-hastype)
      with s have  $t \cdot \sigma = s$  by blast
    } note subst = this
    from subst[OF var] subst[OF f]
    show ?thesis unfolding id by auto
  qed
qed
from arg-cong[OF this, of fst]
have flatten-triv-sort  $t \cdot \sigma = t \cdot \sigma$  unfolding flatten-triv-sort-def by simp
}
thus flatten-triv-sort  $\iota$  eqc  $\cdot_{set} \sigma = eqc \cdot_{set} \sigma$ 
by (metis (no-types, lifting) image-cong image-image)
qed
show finite-constructor-form-pat (flatten-triv-sort-pat p)
unfolding flatten-triv-sort-pat-def ball-simps finite-constructor-form-defs
proof (intro ballI conjI)
  fix mp eqc
  assume  $mp \in p$   $eqc \in mp$ 
  from assms[unfolded finite-constructor-form-defs, rule-format, OF this] obtain
 $\iota$ 
    where fin: finite-sort  $C \iota$  and  $t : \bigwedge t. t \in eqc \implies t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \iota SS)$ 
    and ne:  $eqc \neq \{\}$  by auto
  show flatten-triv-sort  $\iota$  eqc  $\neq \{\}$  using ne by auto
  show  $\exists \iota. finite-sort C \iota \wedge (\forall t \in eqc. flatten-triv-sort t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \iota SS))$ 
  proof (intro exI[of -] conjI fin ballI)
    fix t
    assume  $t \in eqc$ 
    from t[OF this] have  $t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \iota SS)$  by auto
    hence fst (flatten-triv-sort-main  $t$ ) :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \iota SS) \wedge snd$  (flatten-triv-sort-main
 $t$ ) =  $\iota$ 
    proof (induct)
      case (Var x  $\iota$ )

```

```

thus ?case by (auto simp: hastype-def restrict-map-def split: if-splits)
next
  case (Fun f ts  $\iota$  s)
  from Fun(1) have C: C (f,  $\iota$ s) = Some  $\iota$  by (rule fun-hastypeD)
  from C-sub-S[OF Fun(1)] have  $\iota$ :  $\iota \in S$  by auto
  from Fun(3) have map-snd: map snd (map flatten-triv-sort-main ts) =  $\iota$ s
  apply (intro nth-equalityI)
  subgoal by (auto simp: list-all2-conv-all-nth)
  subgoal for i using list-all2-nthD[OF Fun(3), of i] by auto
  done
show ?case
proof (cases cd-sort  $\iota = 1$ )
  case False
  hence id: flatten-triv-sort-main (Fun f ts) =
    (Fun f (map fst (map flatten-triv-sort-main ts)),  $\iota$ )
  using map-snd C by simp
  show ?thesis unfolding id fst-conv snd-conv
  apply (intro conjI refl Fun-hastypeI[OF Fun(1)] list-all2-all-nthI)
  subgoal using Fun(3) by (auto simp: list-all2-conv-all-nth)
  subgoal for i using list-all2-nthD[OF Fun(3), of i] by auto
  done
next
  case True
  hence id: flatten-triv-sort-main (Fun f ts) = (Var (0, $\iota$ ),  $\iota$ )
  using C map-snd by auto
  thus ?thesis using  $\iota$  by (simp add: hastype-def restrict-map-def)
qed
qed
thus flatten-triv-sort t :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid SS)$  unfolding flatten-triv-sort-def by
auto
qed
qed
qed

```

lemma eliminate-uniq-spp: **assumes** finite-constructor-form-pat p
and p = insert mp p'
and mp = insert eqc mp'
and pn = insert mp' p'
and UNIQ eqc
shows simple-pat-complete C SS p \longleftrightarrow simple-pat-complete C SS pn
finite-constructor-form-pat pn
proof
show simple-pat-complete C SS p \implies simple-pat-complete C SS pn **unfolding**
assms
by (auto simp: simple-pat-complete-def simple-match-complete-wrt-def)
show finite-constructor-form-pat pn
using assms **unfolding** finite-constructor-form-defs **by** auto
from \langle UNIQ eqc \rangle **have** UNIQ-subst σ eqc **for** $\sigma :: - \Rightarrow ('f, \text{unit})\text{term}$
unfolding UNIQ-subst-def **by** (rule image-Uniq)

thus *simple-pat-complete* C SS $pn \implies$ *simple-pat-complete* C SS p **unfolding**
assms
by (*auto simp: simple-pat-complete-def simple-match-complete-wrt-def*)
qed

lemma *decompose-spp: assumes finite-constructor-form-pat p*
and $p: p = \text{insert } mp \ p'$
and $mp: mp = \text{insert } eqc \ mp'$
and $root: \bigwedge t. t \in eqc \implies \text{root } t = \text{Some } (f, n)$
and $eqcn: eqcn = (\lambda i. (\lambda t. \text{args } t ! i) \text{ ' } eqc) \text{ ' } \{0..<n\}$
and $pn: pn = (\text{if } \text{Ball } eqcn \ (\lambda eq. \text{UNIQ } (\mathcal{T}(C, \mathcal{V}) \text{ ' } eq)) \text{ then } \text{insert } (eqcn \cup \ mp') \ p' \text{ else } p')$
shows *simple-pat-complete* C SS $p \longleftrightarrow$ *simple-pat-complete* C SS pn
finite-constructor-form-pat pn
proof –
{
 fix t
 assume $t \in eqc$
 from $root$ [*OF this*]
 have $t = \text{Fun } f \ (\text{map } (\lambda i. \text{args } t ! i) \ [0..<n])$
 by (*cases t; auto intro: nth-equalityI*)
} **note** *to-args = this*
from $assms(1)$ [*unfolded finite-constructor-form-pat-def p*]
have *finite-constructor-form-mp mp* **by** *auto*
from $this$ [*unfolded mp finite-constructor-form-mp-def*]
obtain ι **where** *typed: $\bigwedge t. t \in eqc \implies t : \iota$ in $\mathcal{T}(C, \mathcal{V})$ | ' SS* **by** *auto*
have *simple-pat-complete* C SS $p \longleftrightarrow$ *simple-pat-complete* C SS ($\text{insert } (eqcn \cup \ mp') \ p'$)
 unfolding *simple-pat-complete-def p pn* *beq-simps mp* *simple-match-complete-wrt-def*
ball-simps ball-Un
proof (*intro all-cong disj-cong refl conj-cong*)
 fix $\sigma :: - \Rightarrow (-, \text{unit}) \text{term}$
 show $\text{UNIQ-subst } \sigma \ eqc = \text{Ball } eqcn \ (\text{UNIQ-subst } \sigma)$
proof
 assume $uniq: \text{UNIQ-subst } \sigma \ eqc$
 show $\text{Ball } eqcn \ (\text{UNIQ-subst } \sigma)$ **unfolding** $eqcn$
proof *clarsimp*
 fix i
 assume $i: i < n$
 show $\text{UNIQ-subst } \sigma \ \{\text{args } t ! i \mid t \in eqc\}$ (**is** $\text{UNIQ-subst } - \ ?\text{args}$)
 proof (*intro UNIQ-subst-pairI*)
 fix $si \ ti$
 assume $*$: $si \in \ ?\text{args } ti \in \ ?\text{args}$
 from $*$ **obtain** s **where** $s: s \in eqc$ **and** $si: si = \text{args } s ! i$ **by** *auto*
 from $*$ **obtain** t **where** $t: t \in eqc$ **and** $ti: ti = \text{args } t ! i$ **by** *auto*
 from arg-cong [*OF UNIQ-subst-pairD* [*OF uniq s t*], *of $\lambda t. \text{args } t ! i$*] i
 show $si \cdot \sigma = ti \cdot \sigma$ **unfolding** $si \ ti$ **using** $root$ [*OF s*] $root$ [*OF t*] **by** (*cases*
 $s; \text{cases } t; \text{auto}$)

```

    qed
  qed
next
  assume Ball eqcn (UNIQ-subst  $\sigma$ )
  from this[unfolded eqcn, simplified]
  have uniq:  $i < n \implies \text{UNIQ-subst } \sigma \{ \text{args } t ! i \mid t \in \text{eqc} \}$  for i by auto
  show UNIQ-subst  $\sigma$  eqc
  proof (intro UNIQ-subst-pairI)
    fix s t
    assume s:  $s \in \text{eqc}$  and t:  $t \in \text{eqc}$ 
    show  $s \cdot \sigma = t \cdot \sigma$ 
      apply (subst to-args[OF s], subst to-args[OF t])
      apply clarsimp
      subgoal for i using UNIQ-subst-pairD[OF uniq[of i]] s t by auto
      done
    qed
  qed
  also have  $\dots = \text{simple-pat-complete } C \text{ } SS \text{ } pn$ 
  proof (cases  $\forall \text{eq} \in \text{eqcn. UNIQ } (\mathcal{T}(C, \mathcal{V}) \text{ ' } \text{eq})$ )
    case True
      thus ?thesis unfolding pn by auto
    next
      case False
        hence pn:  $pn = p'$  unfolding pn by auto
        from False obtain eq where eq-mem:  $\text{eq} \in \text{eqcn}$  and nuniq:  $\neg \text{UNIQ } (\mathcal{T}(C, \mathcal{V}) \text{ ' } \text{eq})$ 
        'eq by auto
        from nuniq[unfolded Uniq-def] obtain si ti where
          st:  $si \in \text{eq} \text{ } ti \in \text{eq}$  and diff:  $\mathcal{T}(C, \mathcal{V}) \text{ } si \neq \mathcal{T}(C, \mathcal{V}) \text{ } ti$  by auto
        from eq-mem[unfolded eqcn] obtain i where  $i < n$  and eq:  $\text{eq} = \{ \text{args } t ! i \mid t \in \text{eqc} \}$  by auto
        {
          fix ti
          assume ti  $\in \text{eq}$ 
          from this[unfolded eq] obtain t where ti:  $ti = \text{args } t ! i$  and t:  $t \in \text{eqc}$  by
auto
          from typed[OF t] have  $t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \text{' } SS)$  by auto
          with to-args[OF t] have Fun f (map ( $!$ ) (args t)) [ $0..<n$ ]:  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \text{' } SS)$  by auto
          from this[unfolded Fun-hastype list-all2-conv-all-nth] i
          have  $\exists \iota. ti : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \text{' } SS)$  unfolding ti by force
        }
        from this[OF st(1)] this[OF st(2)] obtain vs vt where
          typed:  $si : vs$  in  $\mathcal{T}(C, \mathcal{V} \mid \text{' } SS)$   $ti : vt$  in  $\mathcal{T}(C, \mathcal{V} \mid \text{' } SS)$  by auto
        hence  $si : vs$  in  $\mathcal{T}(C, \mathcal{V})$   $ti : vt$  in  $\mathcal{T}(C, \mathcal{V})$ 
        by (meson hastype-in-Term-mono-right restrict-submap)+
        with diff have diff:  $vs \neq vt$  unfolding hastype-def by auto
        show ?thesis unfolding simple-pat-complete-def pn
        proof (intro all-cong)

```

```

fix  $\sigma$ 
assume sig:  $\sigma :_s \mathcal{V} \mid ' SS \rightarrow \mathcal{T}(C)$ 
have  $\neg$  simple-match-complete-wrt  $\sigma$  (eqcn  $\cup$  mp')
proof
  assume simple-match-complete-wrt  $\sigma$  (eqcn  $\cup$  mp')
  hence simple-match-complete-wrt  $\sigma$  eqcn unfolding simple-match-complete-wrt-def
by auto
  from this[unfolded simple-match-complete-wrt-def] eq-mem
  have UNIQ-subst  $\sigma$  eq by auto
  with st have equiv:  $si \cdot \sigma = ti \cdot \sigma$  by (metis UNIQ-subst-pairD)
  from typed sig
  have typed:  $si \cdot \sigma : \iota_s$  in  $\mathcal{T}(C)$   $ti \cdot \sigma : \iota_t$  in  $\mathcal{T}(C)$  by (metis subst-hastype)+
  hence  $si \cdot \sigma \neq ti \cdot \sigma$  using diff unfolding hastype-def by auto
  with equiv show False by simp
qed
  thus Bex (insert (eqcn  $\cup$  mp') p') (simple-match-complete-wrt  $\sigma$ ) = Bex p'
(simple-match-complete-wrt  $\sigma$ )
  by simp
qed
qed
finally show simple-pat-complete C SS p = simple-pat-complete C SS pn .
show finite-constructor-form-pat pn using assms(1)
  unfolding pn p mp finite-constructor-form-pat-def ball-simps
proof (simp, intro impI)
  assume fin: finite-constructor-form-mp (insert eqc mp')  $\wedge$  Ball p' finite-constructor-form-mp

  and uniq:  $\forall eq \in eqcn. UNIQ (\mathcal{T}(C, \mathcal{V}) ' eq)$ 
show finite-constructor-form-mp (eqcn  $\cup$  mp') unfolding finite-constructor-form-mp-def
proof
  fix eqc'
  assume eqc'  $\in$  eqcn  $\cup$  mp'
  thus eqc'  $\neq \{\}$   $\wedge$  ( $\exists \iota. finite-sort C \iota \wedge (\forall t \in eqc'. t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid ' SS)$ ))
proof
  assume eqc'  $\in$  mp'
  thus ?thesis using fin unfolding finite-constructor-form-mp-def by auto
next
  assume eqc'-mem: eqc'  $\in$  eqcn
  from this[unfolded eqcn] obtain i where i:  $i < n$  and
    eqc': eqc' = {args t ! i | . t  $\in$  eqc} by auto
  from fin[unfolded finite-constructor-form-mp-def] obtain  $\iota$ 
    where fin: finite-sort C  $\iota$  and ne: eqc  $\neq \{\}$  and typed:  $t \in eqc \implies t : \iota$ 
in  $\mathcal{T}(C, \mathcal{V} \mid ' SS)$  for t
  by auto
  from ne obtain t where t: t  $\in$  eqc by auto
  from typed[OF t] have t :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid ' SS)$  by auto
  with to-args[OF t]
  have Fun f (map (!) (args t)) [0.. $n$ ] :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid ' SS)$  by auto
  from finite-arg-sorts[OF fin this, of args t ! i] i
  obtain  $\iota_i$  where  $ti: args t ! i : \iota_i$  in  $\mathcal{T}(C, \mathcal{V} \mid ' SS)$  and fin: finite-sort C

```

ιi **by** *auto*
have *ti-mem*: $\text{args } t ! i \in \text{eqc}'$ **unfolding** *eqc'* **using** $i t$ **by** *auto*
have *wti*: $\text{args } t ! i : \iota i$ **in** $\mathcal{T}(C, \mathcal{V})$ **using** *ti*
by (*meson hastype-in-Term-mono-right restrict-submap*)
from *ti-mem ti wti* **obtain** *ti* **where**
 $ti: ti : \iota i$ **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ $ti : \iota i$ **in** $\mathcal{T}(C, \mathcal{V})$ $ti \in \text{eqc}'$ **by** *auto*
show *?thesis*
proof (*intro exI[of - ιi] conjI ballI fin*)
show $\text{eqc}' \neq \{\}$ **using** *ne* **unfolding** *eqc'* **by** *auto*
fix *si*
assume *si*: $si \in \text{eqc}'$
with *uniq*[*rule-format, OF eqc'-mem*] *ti*
have *type-si*: $si : \iota i$ **in** $\mathcal{T}(C, \mathcal{V})$
unfolding *Uniq-def hastype-def* **by** *auto*
from *si*[*unfolded eqc'*] **obtain** *s* **where** *si*: $si = \text{args } s ! i$ **and** $s : s \in \text{eqc}$
by *auto*
from *typed*[*OF s*] *to-args*[*OF s*]
have *Fun f* (*map* ($!$) (*args s*)) [$0..<n$] : ι **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ **by** *auto*
from *this*[*unfolded Fun-hastype list-all2-conv-all-nth*] *i si*
have $\exists \iota i. si : \iota i$ **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ **by** *auto*
then obtain $\iota 2$ **where** *si*: $si : \iota 2$ **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ **by** *auto*
hence $si : \iota 2$ **in** $\mathcal{T}(C, \mathcal{V})$
by (*meson hastype-in-Term-mono-right restrict-submap*)
with *type-si* **have** $\iota i = \iota 2$ **by** (*auto simp: hastype-def*)
thus $si : \iota i$ **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ **using** *si* **by** *auto*
qed
qed
qed
qed
qed
qed
lemma *eliminate-clash-spp*: **assumes** *finite-constructor-form-pat p*
and $p = \text{insert } mp \ pn$
and $mp = \text{insert } \text{eqc } mp'$
and $\text{eqc} = \{s, t\} \cup \text{eqc}'$
and *Conflict-Clash s t*
shows *simple-pat-complete C SS p* \longleftrightarrow *simple-pat-complete C SS pn*
finite-constructor-form-pat pn
proof
from *mp-fail-pcorrect1* [*OF match-fail-mp-fail*] [*OF match-fail.match-clash'*] [*OF assms(5)*],
of - undefined {#} λ -. None, unfolded match-complete-wrt-def tvar-match-def,
simplified
have *clash*: $\sigma :_s \mathcal{V} \mid '(vars s \cup vars t) \rightarrow \mathcal{T}(C) \implies s \cdot \sigma \neq t \cdot \sigma$ **for** σ **by** *metis*
from *assms(1)*[*unfolded finite-constructor-form-pat-def finite-constructor-form-mp-def*]
assms
obtain ι **where** *typed*: $u \in \{s, t\} \implies u : \iota$ **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ **for** u **by** *auto*
have $vars u \subseteq \text{dom } (\mathcal{V} \mid 'SS)$ **if** $u \in \{s, t\}$ **for** u **using** *typed*[*OF that*]
by (*rule hastype-in-Term-imp-vars-subset*)

```

hence  $st: \text{vars } s \cup \text{vars } t \subseteq SS$  by auto
{
  fix  $\sigma$ 
  assume  $\text{sig}: \sigma :_s \mathcal{V} \mid' SS \rightarrow \mathcal{T}(C)$ 
  have  $s \cdot \sigma \neq t \cdot \sigma$ 
  proof (rule clash)
    show  $\sigma :_s \mathcal{V} \mid' (\text{vars } s \cup \text{vars } t) \rightarrow \mathcal{T}(C)$ 
    using  $\text{sig } st$  by (meson restrict-map-mono-right sorted-map-cmono)
  qed
  hence  $\neg \text{UNIQ-subst } \sigma \text{ eqc unfolding } \text{assms UNIQ-subst-def Uniq-def}$  by auto
}
thus  $\text{simple-pat-complete } C \text{ SS } p \implies \text{simple-pat-complete } C \text{ SS } pn$  unfolding
assms(2-3)
by (auto simp: simple-pat-complete-def simple-match-complete-wrt-def)
show finite-constructor-form-pat  $pn$ 
using  $\text{assms unfolding finite-constructor-form-defs}$  by auto
show  $\text{simple-pat-complete } C \text{ SS } pn \implies \text{simple-pat-complete } C \text{ SS } p$  unfolding
assms
by (auto simp: simple-pat-complete-def simple-match-complete-wrt-def)
qed

```

```

lemma detect-sat-spp:
  assumes  $p = \text{insert } \{\} p'$ 
  shows  $\text{simple-pat-complete } C \text{ SS } p$ 
  unfolding  $\text{assms}$  by (auto simp: simple-pat-complete-def simple-match-complete-wrt-def)

```

```

lemma detect-unsat-spp:
  shows  $\neg \text{simple-pat-complete } C \text{ SS } \{\}$ 
  using  $\sigma g'$  by (auto simp: simple-pat-complete-def simple-match-complete-wrt-def)

```

```

lemma separate-var-fun-spp: assumes finite-constructor-form-pat  $p$ 
  and  $p: p = \text{insert } mp \ p'$ 
  and  $mp: mp = \text{insert } \text{eqc } mp'$ 
  and  $\text{eqc}: \text{eqc} = \{x,t\} \cup \text{eqc}V \cup \text{eqc}F$ 
  and  $Pn: Pn = \{\text{insert } \{\{x,t\}\} p', \text{insert } (\{\text{insert } x \ \text{eqc}V\}) p', \text{insert } (\{\text{insert } t \ \text{eqc}F\} \cup mp') p'\}$ 
  shows  $\text{simple-pat-complete } C \text{ SS } p \iff (\forall pn \in Pn. \text{simple-pat-complete } C \text{ SS } pn)$ 

```

```

  Ball Pn finite-constructor-form-pat
proof –
  let  $?spc = \text{simple-pat-complete } C \text{ SS}$ 
  note  $\text{spc} = \text{simple-pat-complete-def simple-match-complete-wrt-def}$ 
  have  $\text{main}: \text{UNIQ-subst } \sigma \ \text{eqc} \iff \text{UNIQ-subst } \sigma \ \{x,t\} \wedge \text{UNIQ-subst } \sigma \ (\text{insert } x \ \text{eqc}V) \wedge \text{UNIQ-subst } \sigma \ (\text{insert } t \ \text{eqc}F)$ 
  (is  $?l = ?r$ ) for  $\sigma :: - \Rightarrow (-, \text{unit})\text{term}$ 
  proof
    assume  $?l$ 

```

```

show ?r
  by (intro conjI; rule UNIQ-subst-mono[OF - ⟨?l⟩], auto simp: eqc)
next
  assume ?r
  hence *: UNIQ-subst  $\sigma$  {x, t} UNIQ-subst  $\sigma$  (insert x eqcV) UNIQ-subst  $\sigma$ 
(insert t eqcF) by auto
  show ?l
  proof (intro UNIQ-subst-pairI)
    fix a b
    assume a ∈ eqc b ∈ eqc
    with
      UNIQ-subst-pairD[OF *(1), of x t]
      UNIQ-subst-pairD[OF *(2), of x]
      UNIQ-subst-pairD[OF *(2), of a b]
      UNIQ-subst-pairD[OF *(3), of t]
      UNIQ-subst-pairD[OF *(3), of a b]
    show a ·  $\sigma$  = b ·  $\sigma$  unfolding eqc by fastforce
  qed
qed
have ?spc p  $\longleftrightarrow$  (?spc (insert {eqc} p') ∧ ?spc (insert mp' p'))
  unfolding p mp spc by auto
also have ...  $\longleftrightarrow$ 
  ?spc (insert {{x,t}} p') ∧ (?spc (insert mp' p') ∧ ?spc (insert {insert x eqcV}
p') ∧ ?spc (insert {insert t eqcF} p'))
  (is -  $\longleftrightarrow$  - ∧ ?sub)
  unfolding spc by (auto simp: main)
  also have ?sub  $\longleftrightarrow$  ?spc (insert ({insert x eqcV}) p') ∧ ?spc (insert ({insert t
eqcF} ∪ mp') p')
  by (auto simp: spc)
  finally show ?spc p  $\longleftrightarrow$  (∀ pn ∈ Pn. ?spc pn) unfolding Pn by auto
  show Ball Pn finite-constructor-form-pat using assms(1) unfolding assms
  by (auto simp: finite-constructor-form-pat-def finite-constructor-form-mp-def)
qed

```

```

lemma separate-var-fun-spp-single: assumes finite-constructor-form-pat p
  and p: p = insert mp p'
  and mp: mp = insert eqc mp'
  and eqc: eqc = {x,t} ∪ eqc'
  and Pn: Pn = {insert {{x,t}} p', insert ({insert x eqc'} ∪ mp') p'}
shows simple-pat-complete C SS p  $\longleftrightarrow$  (∀ pn ∈ Pn. simple-pat-complete C SS pn)

```

```

  Ball Pn finite-constructor-form-pat
proof –
  let ?spc = simple-pat-complete C SS
  have swap: {t, x} = {x,t} by auto
  from eqc have eqc: eqc = {t,x} ∪ {} ∪ eqc' by auto
  note step = separate-var-fun-spp[OF assms(1) p mp eqc refl]
  from step(2) show Ball Pn finite-constructor-form-pat unfolding Pn swap by
auto

```

show *simple-pat-complete* C SS $p \iff (\forall pn \in Pn. \text{simple-pat-complete } C \text{ } SS \text{ } pn)$
unfolding *step(1)* **unfolding** *swap* Pn
by (*simp*, *auto simp: simple-pat-complete-def simple-match-complete-wrt-def*)
qed

lemma *instantiate-spp: assumes finite-constructor-form-pat* p
and *disj: fst ' tvars-spat* $p \cap \{n..<n + m\} = \{\}$
and *x: x ∈ tvars-spat* p
and *Pn: Pn = (λ τ. (⋅) ((⋅) (λ t. t ⋅ τ)) ' p) ' τ s n x*
shows *simple-pat-complete* C SS $p \iff (\forall pn \in Pn. \text{simple-pat-complete } C \text{ } SS \text{ } pn)$

Ball Pn finite-constructor-form-pat

proof

assume *comp: simple-pat-complete* C SS p

show $(\forall pn \in Pn. \text{simple-pat-complete } C \text{ } SS \text{ } pn)$

unfolding *simple-pat-complete-def*

proof (*intro ballI allI impI*)

fix $pn \sigma$

assume $pn \in Pn$ **and** *sig: σ :_s V | ' SS → T(C)*

from *this[unfolded Pn]* **obtain** τ **where** *tau: τ ∈ τ s n x*

and *pn: pn = (⋅) ((⋅) (λ t. t ⋅ τ)) ' p* **by** *auto*

from *tau[unfolded τ s-def]* **obtain** $f \ \iota s$ **where**

tau: τ = τ c n x (f, ι s) **and** *f: f : ι s → snd x in C* **by** *auto*

define t **where** $t = \text{Fun } f \ (\text{map } (\lambda i. \sigma \ (n + i, \iota s ! i)) \ [0 \ ..< \ \text{length } \iota s])$

from *C-sub-S[OF f]* **have** $x: \text{snd } x \in S$ **and** $\iota s: \text{set } \iota s \subseteq S$ **by** *auto*

have $t: t : \text{snd } x \text{ in } \mathcal{T}(C)$

unfolding *t-def*

proof (*intro Fun-hastypeI[OF f] list-all2-all-nthI, force, clarsimp*)

fix i

assume $i < \text{length } \iota s$

hence $\iota s ! i \in S$ **using** ιs **by** *auto*

with *sig* **show** $\sigma \ (n + i, \iota s ! i) : \iota s ! i \text{ in } \mathcal{T}(C)$

by (*simp add: restrict-map-eq-restrict-sset sorted-map-def*)

qed

define σ' **where** $\sigma' = \sigma(x := t)$

from *sig t* **have** *sig': σ' :_s V | ' SS → T(C)* **unfolding** *σ'-def sorted-map-def*

by (*metis comp-apply fun-upd-apply hastypeD hastypeI hastype-restrict*)

from *comp[unfolded simple-pat-complete-def, rule-format, OF this]*

obtain mp **where** $mp: mp \in p$ **and** *comp: simple-match-complete-wrt σ' mp*

by *auto*

let $?mp = (\cdot) (\lambda t. t \cdot \tau) ' mp$

from mp **have** $mem: ?mp \in pn$ **unfolding** pn **by** *auto*

{

fix i

assume $i < \text{length } \iota s$ **and**

$x: x = (n + i, \iota s ! i)$

```

from  $m[OF\ f]$   $i$  have  $i < m$  by auto
with  $disj\ x\ assms(\mathcal{J})$  have False by fastforce
} note  $x-disj = this$ 

have  $x\tauau: x \notin vars\ (s \cdot \tau)$  for  $s$ 
proof
  assume  $x \in vars\ (s \cdot \tau)$ 
  from  $this[unfolding\ tau\ \tau c-def\ split\ vars-term-subst\ subst-def]$ 
  obtain  $i$  where  $i: i < length\ \iota s$  and
     $x: x = (n + i, \iota s\ !\ i)$  by (auto simp: set-conv-nth nth-indexed-from-eq split:
if-splits)
  with  $x-disj$  show False by auto
qed

show  $Bex\ pn$  (simple-match-complete-wrt  $\sigma$ )
proof (intro  $beXI[OF - mem]$ )
  have simple-match-complete-wrt  $\sigma\ ?mp = simple-match-complete-wrt\ \sigma'\ ?mp$ 

  proof (rule  $tvars-spat-cong[of\ pn]$ )
    show  $?mp \in pn$  unfolding  $pn$  using  $mp$  by auto
    fix  $y$ 
    assume  $y \in tvars-spat\ pn$ 
    thus  $\sigma\ y = \sigma'\ y$ 
    unfolding  $\sigma'-def\ pn$  by (auto simp: xtau)
  qed
  also have ...
    unfolding simple-match-complete-wrt-def
  proof (intro  $ballI\ UNIQ-subst-pairI, clarsimp$ )
    fix  $eqc\ s\ s'$ 
    assume  $eqc: eqc \in mp$  and  $st: s \in eqc\ s' \in eqc$ 
    from  $comp[unfolding\ simple-match-complete-wrt-def\ UNIQ-subst-alt-def,$ 
rule-format, OF this]
    have  $eq: s \cdot \sigma' = s' \cdot \sigma'$  .
    {
      fix  $y$ 
      have  $(\tau \circ_s \sigma')\ y = \sigma'\ y$ 
      proof (cases  $y = x$ )
        case False
        thus  $?thesis$  unfolding subst-compose-def tau tau c-def split by (simp add:
subst-def)
      next
        case True
        show  $?thesis$  unfolding True using  $x-disj$ 
        by (auto simp add: tau tau c-def subst-compose-def sigma'-def t-def
nth-indexed-from-eq intro!: nth-equalityI)
      qed
    }
    thus  $s \cdot \tau \cdot \sigma' = s' \cdot \tau \cdot \sigma'$  using  $eq$ 
    by (metis subst-same-vars subst-subst-compose)

```

```

      qed
      finally show simple-match-complete-wrt  $\sigma$  ?mp by auto
    qed
  qed
next
{
  from  $x$  obtain eqc mp t where *: eqc  $\in$  mp  $mp \in p$   $t \in eqc$  and  $x: x \in vars$ 
  t by auto
  from * assms(1)[unfolded finite-constructor-form-pat-def]
  have finite-constructor-form-mp mp by auto
  from this[unfolded finite-constructor-form-mp-def] * obtain  $\iota$ 
    where  $t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid^{\iota} SS)$  by auto
  from hastype-in-Term-imp-vars[OF this x]
  have  $snd\ x \in S$  unfolding restrict-map-def by (auto split: if-splits)
} note  $xS = this$ 

assume comp:  $\forall pn \in Pn. simple-pat-complete\ C\ SS\ pn$ 
show simple-pat-complete  $C\ SS\ p$  unfolding simple-pat-complete-def
proof (intro impI allI)
  fix  $\sigma$ 
  assume sig:  $\sigma :_s \mathcal{V} \mid^{\iota} SS \rightarrow \mathcal{T}(C)$ 
  have  $x : snd\ x$  in  $\mathcal{V} \mid^{\iota} SS$  using  $xS$  by (cases x, auto simp: hastype-def
restrict-map-def)
  with sig have sigx:  $\sigma\ x : snd\ x$  in  $\mathcal{T}(C)$  by (rule sorted-mapD)
  then obtain  $f\ ts$  where sigxF:  $\sigma\ x = Fun\ f\ ts$ 
    by (cases  $\sigma\ x$ , auto)
  from sigx[unfolded this Fun-hastype] obtain  $\iota_s$ 
    where  $f: f : \iota_s \rightarrow snd\ x$  in  $C$  and  $ts: ts :_{\iota} \iota_s$  in  $\mathcal{T}(C)$  by auto
  define  $\tau$  where  $\tau = \tau c\ n\ x\ (f, \iota_s)$ 
  define cond where cond  $y = (fst\ y \in \{n..<n+length\ \iota_s\} \wedge snd\ y = \iota_s ! (fst\ y$ 
  -  $n))$  for  $y$ 
  define  $\sigma'$  where  $\sigma'\ y = (if\ cond\ y\ then\ ts\ !\ (fst\ y - n)\ else\ \sigma\ y)$  for  $y$ 
  {
    fix  $y$ 
    assume cond  $y$ 
    hence  $\exists i. y = (n + i, \iota_s ! i) \wedge \sigma'\ y = ts ! i \wedge i < length\ \iota_s$ 
    unfolding cond-def sigma'-def by (cases y, auto intro!: exI[of - fst y - n])
  } note cond = this
  have sig':  $\sigma' :_s \mathcal{V} \mid^{\iota} SS \rightarrow \mathcal{T}(C)$ 
proof
  fix  $y\ \iota$ 
  assume  $y: y : \iota$  in  $\mathcal{V} \mid^{\iota} SS$ 
  show  $\sigma'\ y : \iota$  in  $\mathcal{T}(C)$ 
  proof (cases cond y)
    case False
    hence  $\sigma'\ y = \sigma\ y$  unfolding sigma'-def by auto
    with  $y\ sig$  show ?thesis by (metis sorted-mapE)
  next
  case True

```

```

    from cond[OF this]
    obtain i where *: i < length  $\iota s$   $\sigma' y = \iota s ! i y = (n + i, \iota s ! i)$  by auto
    from  $*(\beta) y$  have  $\iota = \iota s ! i$ 
      by (auto simp: hastype-def restrict-map-def split: if-splits)
    with  $y$  * show ?thesis using list-all2-nthD2[OF  $\iota s$ , of  $i$ ] by auto
  qed
  qed
  let ?p = ( $\cdot$ ) (( $\cdot$ ) ( $\lambda t. t \cdot \tau$ )) ' $p$ 
  from  $f$  have  $\tau \in \tau s n x$  unfolding  $\tau s$ -def  $\tau$ -def by auto
  hence ?p  $\in Pn$  by (auto simp: assms)
  from comp[rule-format, OF this, unfolded simple-pat-complete-def, rule-format,
  OF sig^]
  obtain  $mp$  where  $mp: mp \in p$  and  $comp: simple-match-complete-wrt \sigma' ((\cdot)$ 
  ( $\lambda t. t \cdot \tau$ ) ' $mp$ ) by auto
  show Bex  $p$  (simple-match-complete-wrt  $\sigma$ )
  proof (intro bexI[OF -  $mp$ ])
    have simple-match-complete-wrt  $\sigma mp = simple-match-complete-wrt \sigma' mp$ 
    proof (rule tvars-spat-cong[OF -  $mp$ ])
      fix  $y$ 
      assume  $yp: y \in tvars-spat p$ 
      show  $\sigma y = \sigma' y$ 
      proof (cases cond  $y$ )
        case False
        thus ?thesis unfolding  $\sigma'$ -def by auto
      next
        case True
        from cond[OF this] obtain  $i$  where  $y: y = (n + i, \iota s ! i)$  and  $i: i <$ 
        length  $\iota s$  by auto
        from disj  $yp$  have  $fst y \notin \{n ..< n + m\}$  by fastforce
        with  $y$  have  $i \geq m$  by auto
        with  $m$ [OF  $f$ ]  $i$  have False by auto
        thus ?thesis ..
      qed
    qed
  also have ... unfolding simple-match-complete-wrt-def UNIQ-subst-alt-def
  proof clarify
    fix  $eqc s t$ 
    assume  $eqc \in mp s \in eqc t \in eqc$ 
    from comp[unfolded simple-match-complete-wrt-def UNIQ-subst-alt-def,
    rule-format, OF imageI imageI imageI, OF this]
    have  $eq: s \cdot \tau \cdot \sigma' = t \cdot \tau \cdot \sigma'$  .
    {
      fix  $y$ 
      have  $(\tau \circ_s \sigma') y = \sigma' y$ 
      proof (cases  $y = x$ )
        case False
        thus ?thesis unfolding subst-compose-def  $\tau$ -def  $\tau c$ -def split by (simp
        add: subst-def)
      next

```

```

    case True
    have cx:  $\neg$  cond x
    proof
      assume cond x
      from cond[OF this] m[OF f] have fst x  $\in$  {n ..< n + m} by auto
      with disj x show False by blast
    qed
    hence sig'x:  $\sigma'$  x =  $\sigma$  x unfolding  $\sigma'$ -def by auto
    show ?thesis unfolding True sig'x sigxF using cx ts[unfolded list-all2-conv-all-nth]
      by (auto simp add:  $\tau$ c-def subst-compose-def  $\sigma'$ -def  $\tau$ -def cond-def
nth-indexed-from-eq intro!: nth-equalityI)
  }
  with eq show s  $\cdot$   $\sigma'$  = t  $\cdot$   $\sigma'$  by (metis eval-same-vars-cong eval-subst)
  qed
  finally show simple-match-complete-wrt  $\sigma$  mp .
  qed
  qed
next
show Ball Pn finite-constructor-form-pat
  unfolding finite-constructor-form-pat-def finite-constructor-form-mp-def
  proof (intro ballI, goal-cases)
  case (1 pn mpn eqcn)
  from this[unfolded Pn] obtain  $\tau$  mp eqc where
    mem: mp  $\in$  p eqc  $\in$  mp and eqcn: eqcn = ( $\lambda$  t. t  $\cdot$   $\tau$ ) ' eqc
    and tau:  $\tau \in \tau$ s n x
  by auto
  from tau[unfolded  $\tau$ s-def] obtain f  $\sigma$ s where f: f :  $\sigma$ s  $\rightarrow$  snd x in C and tau:
 $\tau = \tau$ c n x (f, $\sigma$ s) by auto
  from assms(1)[unfolded finite-constructor-form-pat-def finite-constructor-form-mp-def,
rule-format, OF mem]
  obtain  $\iota$  where ne: eqc  $\neq$  {} and fin: finite-sort C  $\iota$  and typed: t  $\in$  eqc  $\implies$ 
t :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  for t by auto
  show eqcn  $\neq$  {}  $\wedge$  ( $\exists$   $\iota$ . finite-sort C  $\iota$   $\wedge$  ( $\forall$  t $\in$ eqcn. t :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ ))
  proof (intro conjI exI[of -]  $\iota$  fin ballI)
  show eqcn  $\neq$  {} using ne unfolding eqcn by auto
  fix tt
  assume tt  $\in$  eqcn
  then obtain t where t: t  $\in$  eqc and tt: tt = t  $\cdot$   $\tau$  unfolding eqcn by auto
  from typed[OF t] have t: t :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by auto
  show tt :  $\iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  unfolding tt
  proof (rule subst-hastype[OF - t], standard)
  fix y  $\sigma$ 
  assume y: y :  $\sigma$  in  $\mathcal{V} \mid 'SS$ 
  show  $\tau$  y :  $\sigma$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ 
  proof (cases y = x)
  case False
  with y show ?thesis unfolding tau  $\tau$ c-def split subst-def by simp
  next

```

```

case True
  with y have  $\sigma: \sigma = \text{snd } x$  by (auto simp: hastype-def restrict-map-def
split: if-splits)
  {
    fix i
    assume  $i < \text{length } \sigma s$ 
    hence  $\sigma s ! i \in S$  using C-sub-S[OF f] by auto
    hence  $(n + i, \sigma s ! i) : \sigma s ! i \text{ in } \mathcal{V} \mid 'SS$  by (auto simp: hastype-def
restrict-map-def)
  }
  thus ?thesis unfolding  $\sigma$  True tau  $\tau$ c-def split subst-def
  by (auto simp add: nth-indexed-from-eq intro!: Fun-hastypeI[OF f]
list-all2-all-nthI)
  qed
qed
qed
qed
qed

```

lemma *typed-restrict-imp-typed*: $t : s \text{ in } \mathcal{T}(D, W \mid 'F) \implies t : s \text{ in } \mathcal{T}(D, W)$
by (*meson Term-mono-right restrict-submap subsssetD*)

lemma *eliminate-large-sort*: **assumes**

cond: $\bigwedge i. i \leq (n :: \text{nat}) \implies \text{snd } (x \ i) = \iota \wedge \text{eq } i \in \text{mp } i \wedge \text{Var } (x \ i) \neq t \ i \wedge$
 $\{ \text{Var } (x \ i), t \ i \} \subseteq \text{eq } i$ **and**

vars: $x \ ' \{0..n\} \cap \text{tvars-spat } p = \{ \}$ **and**

large: $\text{card } (t \ ' \{0..n\}) < \text{card-of-sort } C \ \iota$ **and**

fin: *finite-constructor-form-pat* $(p \cup \text{mp } ' \{0..n\})$

shows *simple-pat-complete* $C \ SS \ (p \cup \text{mp } ' \{0..n\}) = \text{simple-pat-complete } C \ SS \ p$

proof

assume *comp*: *simple-pat-complete* $C \ SS \ p$

thus *simple-pat-complete* $C \ SS \ (p \cup \text{mp } ' \{0..n\})$

unfolding *simple-pat-complete-def* **by** *blast*

next

let $?p' = p \cup \text{mp } ' \{0..n\}$

assume *comp*: *simple-pat-complete* $C \ SS \ ?p'$

show *simple-pat-complete* $C \ SS \ p$

unfolding *simple-pat-complete-def*

proof (*intro allI impI, goal-cases*)

case $\sigma: (1 \ \sigma)$

let $?T = t \ ' \{0..n\}$

let $?X = x \ ' \{0..n\}$

let $?TX = ?T \cup (\text{Var } o \ x) \ ' \{0..n\}$

define *Tf* **where** $Tf = (?T - \text{Var } ' \ ?X)$

define *DomT* **where** $\text{DomT} = \{x. \text{Var } x \in ?T \wedge \text{Var } x \notin Tf\}$

have *DomT-alt*: $\text{DomT} = ?X \cap \{x. \text{Var } x \in ?T\}$ **unfolding** *DomT-def Tf-def*

by *auto*

define *Dom* **where** $\text{Dom} = ?X$

define *Avoid* **where** $\text{Avoid} = \{t \cdot \sigma \mid t. t \in Tf\}$

```

define Ran where Ran = {t. t : ι in T(C)} - Avoid
{
  fix i
  assume i: i ≤ n
  note cond = cond[OF this]
  from cond have eq: eq i ∈ mp i by auto
  from i fin have finite-constructor-form-mp (mp i)
    unfolding finite-constructor-form-pat-def by auto
  from this[unfolded finite-constructor-form-mp-def, rule-format, OF eq]
  obtain ι' where fin: finite-sort C ι' and sort: ∧ s. s ∈ eq i ⇒ s : ι' in T(C, V
| 'SS)
  by auto
  from cond have terms: Var (x i) ∈ eq i t i ∈ eq i x i : ι in V by auto
  from sort[OF this(1)] this(3) have id: ι' = ι
    by (metis Var-hastype typed-S-eq)
  note fin = fin[unfolded id]
  note sort = sort[unfolded id]
  from sort[OF terms(1)] have iota: ι ∈ S by (rule typed-imp-S)
  note sort[OF terms(1)] sort[OF terms(2)] fin iota
} note terms = this

have finι: finite-sort C ι using terms[of 0] by auto
have ι ∈ S using terms[of 0] by auto
note terms = terms(1,2)
{
  fix s
  assume s ∈ ?TX
  with terms have s : ι in T(C, V | 'SS) by auto
} note TX = this
hence T: s ∈ ?T ⇒ s : ι in T(C, V | 'SS) for s by auto

{
  define Terms where Terms = ?T
  define GTerms where GTerms = {s. s : ι in T(C)}
  from finι have finG: finite GTerms unfolding GTerms-def finite-sort-def by
auto
  from large[unfolded cd[OF ι ∈ S]] card-of-sort-def]
  have fin: finite Terms
    and card: card Terms < card GTerms
    by (auto simp: Terms-def GTerms-def)
  let ?Var = Var :: nat × 's ⇒ ('f, nat × 's)term
  have splitTerms: ?Var ' DomT ∪ Tf ⊆ Terms ?Var ' DomT ∩ Tf = {}
    unfolding DomT-def Terms-def Tf-def by auto
  from splitTerms finite-subset[OF - fin]
  have fin': finite (?Var ' DomT) finite Tf
    by auto
  from card-mono[OF fin splitTerms(1)] card card-Un-disjoint[OF fin'] split-
Terms
  have card (?Var ' DomT) + card Tf < card GTerms

```

```

    by auto
  hence  $\text{card } \text{Dom}T < \text{card } G\text{Terms} - \text{card } T\text{f}$  using card-image[of ?Var]
    by simp
  also have  $\dots \leq \text{card } G\text{Terms} - \text{card } (T\text{f} \cdot_{\text{set}} \sigma)$  using fin'(2)
    by (meson card-image-le diff-le-mono2)
  also have  $\dots = \text{card } (G\text{Terms} - (T\text{f} \cdot_{\text{set}} \sigma))$ 
  proof (rule card-Diff-subset[symmetric, OF finite-imageI[OF fin'(2)]])
    {
      fix  $ts$ 
      assume  $ts \in T\text{f} \cdot_{\text{set}} \sigma$ 
      then obtain  $t$  where  $t \in ?T$  and  $ts: ts = t \cdot \sigma$ 
        unfolding Tf-def by auto
      with terms have  $t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by auto
      with  $\sigma$  have  $ts \in G\text{Terms}$  unfolding ts GTerms-def by (simp add: subst-hastype)
    }
    thus  $T\text{f} \cdot_{\text{set}} \sigma \subseteq G\text{Terms}$  by auto
  qed
  also have  $T\text{f} \cdot_{\text{set}} \sigma = \text{Avoid}$  unfolding Avoid-def by auto
  also have  $\text{id}: G\text{Terms} - \text{Avoid} = \text{Ran}$  unfolding Ran-def GTerms-def by
simp
  finally have main:  $\text{card } \text{Dom}T < \text{card } \text{Ran}$  .
  from finG have finR: finite Ran unfolding id[symmetric] by blast
  have finD: finite DomT using finite-imageD[OF fin'(1)] by auto
  from main obtain fresh where fresh:  $\text{fresh} \in \text{Ran}$  by (cases Ran = \{\}, auto)
  define RanT where  $\text{Ran}T = \text{Ran} - \{\text{fresh}\}$ 
  have  $\text{card } \text{Ran}T = \text{card } \text{Ran} - 1$  using fresh finR unfolding RanT-def by
auto
  with main have main:  $\text{card } \text{Dom}T \leq \text{card } \text{Ran}T$  by auto
  from finR have finRT: finite RanT unfolding RanT-def by auto
  from card-le-inj[OF finD finRT main]
  obtain h where  $h: h \in \text{Dom}T \rightarrow \text{Ran}T$  and hinj: inj-on h DomT by auto
  define g where  $g\ y = (\text{if } y \in \text{Dom}T \text{ then } h\ y \text{ else } \text{fresh})$  for  $y$ 
  have  $g\text{Ran}: g \in \text{Dom} \rightarrow \text{Ran}$  unfolding g-def using fresh h RanT-def by
auto
  have gInj: inj-on g DomT unfolding g-def using hinj by (auto simp: inj-on-def)
  {
    fix  $y$ 
    assume  $y \in \text{Dom} - \text{Dom}T$ 
    hence  $g\ y = \text{fresh}$  unfolding g-def by auto
    hence  $g\ y \notin \text{Ran}T$  unfolding RanT-def by auto
    hence  $g\ y \notin g\ ' \text{Dom}T$  using h unfolding g-def by auto
  }
  with gRan gInj have  $\exists g. g \in \text{Dom} \rightarrow \text{Ran} \wedge \text{inj-on } g\ \text{Dom}T \wedge (\forall y \in \text{Dom} - \text{Dom}T. g\ y \notin g\ ' \text{Dom}T)$ 
    by blast
  }
  then obtain  $\delta$  where  $\delta\text{Ran}: \delta \in \text{Dom} \rightarrow \text{Ran}$ 

```

and inj: *inj-on* δ *DomT*
and inj': $y \in \text{Dom} - \text{DomT} \implies \delta y \notin \delta \text{ ' DomT}$ **for** y
by *blast*

```

{
  fix y
  assume y ∈ Dom
  hence Var y ∈ ?TX unfolding DomT-def Dom-def by auto
  from TX[OF this] have snd y = ι
    by (simp add: hastype-restrict)
} note Dom = this

```

define σ' **where** $\sigma' x = (\text{if } x \in \text{Dom} \text{ then } \delta x \text{ else } \sigma x)$ **for** x

```

{
  fix s
  assume s ∈ Tf
  hence s: s ∈ ?T and cond: is-Fun s ∨ the-Var s ∉ Dom
    unfolding Tf-def Dom-def by auto
  have s · σ' = s · σ
  proof (intro term-subst-eq)
    fix y
    assume y: y ∈ vars s
    show σ' y = σ y using cond
    proof (cases s)
      case (Fun f ts)
        with y have s ▷ Var y by fastforce
        then obtain c where c: c ≠ Hole and sc: s = c ⟨ Var y ⟩ by blast
        from T[OF s] have s: s : ι in T(C, V |' SS) .
        with σ have sσ: s · σ : ι in T(C) by (rule subst-hastype)
        have y ∉ Dom
        proof
          assume y ∈ Dom
          hence Var y ∈ ?TX unfolding Dom-def DomT-def by auto
          from TX[OF this]
            have Var y : ι in T(C, V |' SS) .
            hence σy: σ y : ι in T(C) using σ
              by (simp add: sorted-mapD)
            obtain t1 d t2 where id: t1 = s · σ d = c ·c σ t2 = σ y by auto
            from sc have eq: t1 = d ⟨ t2 ⟩
              by (simp add: id)
            from σy sσ have t1: t1 : ι in T(C) t2 : ι in T(C) by (auto simp: id)
            from c have dh: d ≠ Hole by (cases c, auto simp: id)
            from dh have size: size (d ⟨ t ⟩) > size t for t by (rule size-ne-ctxt)
            define stack where stack i = ((λ t. d ⟨ t ⟩)  $\overset{\sim}{\sim}$  i) t2 for i
            from t1[unfolded eq]
          have d: d : ι → ι in C(C, λx. None) by (rule apply-ctxt-hastype-imp-hastype-context)

```

```

{
  fix i
  have stack i : ι in  $\mathcal{T}(C) \wedge \text{size}(\text{stack } i) \geq i$ 
  proof (induct i)
    case 0
    thus ?case by (simp add: stack-def t1)
  next
    case (Suc i)
    have stack (Suc i) = d ⟨stack i⟩ unfolding stack-def by simp
  with Suc d size[of stack i] show ?case by (auto intro: apply-ctxt-hastype)
  qed
} note stack = this
have inf: infinite (range stack)
proof
  assume finite (range stack)
  hence finite (size ' range stack) by auto
  then obtain n where  $\bigwedge i. \text{size}(\text{stack } i) \leq n$ 
    by (meson UNIV-I finite-nat-set-iff-bounded-le image-eqI)
  from this[of Suc n] stack[of Suc n] show False by auto
  qed
from stack have range stack  $\subseteq \{t . t : \iota \text{ in } \mathcal{T}(C)\}$  by auto
with inf have infinite {t. t : ι in  $\mathcal{T}(C)$ } by (metis infinite-super)
with fin
show False unfolding finite-sort-def by blast
qed
thus  $\sigma' y = \sigma y$  unfolding  $\sigma'$ -def by auto
next
case (Var y)
with cond y show ?thesis unfolding  $\sigma'$ -def by auto
qed
} note  $\sigma'\sigma = \text{this}$ 

{
  fix x
  assume x ∈ Dom
  with δRan have δ x ∈ Ran by auto
  from this[unfolded Ran-def]
  have δ x ∈ {t. t : ι in  $\mathcal{T}(C)$ } - Avoid .
} note δ = this

from Dom δ σ have  $\sigma': \sigma' :_s \mathcal{V} \mid ' SS \rightarrow \mathcal{T}(C)$ 
  unfolding  $\sigma'$ -def sorted-map-def restrict-map-def by (auto split: if-splits)
from comp[unfolded simple-pat-complete-def, rule-format, OF this]
obtain mp' where mp: mp' ∈ ?p' and comp: simple-match-complete-wrt  $\sigma'$ 
mp'
  by auto

from mp show ?case

```

```

proof
  assume  $mp'$ :  $mp' \in p$ 
  have simple-match-complete-wrt  $\sigma'$   $mp' = \text{simple-match-complete-wrt } \sigma \text{ } mp'$ 
    unfolding simple-match-complete-wrt-def
  proof (intro ball-cong refl)
    fix eqc
    assume eqc:  $eqc \in mp'$ 
    {
      fix s
      assume s:  $s \in eqc$ 
      define V where  $V = \text{tvars-spat } p$ 
      have  $s \cdot \sigma = s \cdot \sigma'$ 
      proof (rule term-subst-eq)
        fix y
        assume  $y \in \text{vars } s$ 
        with eqc s mp' have  $y \in \text{tvars-spat } p$  by auto
        with vars have  $y \notin \text{Dom}$  unfolding V-def[symmetric] Dom-def by
fastforce
          thus  $\sigma y = \sigma' y$  unfolding  $\sigma'$ -def by auto
        qed
      }
    }
  thus UNIQ-subst  $\sigma'$  eqc = UNIQ-subst  $\sigma$  eqc unfolding UNIQ-subst-alt-def
by auto
  qed
  with comp show ?thesis using mp' by auto
next
  assume  $mp' \in mp \text{ } \{0..n\}$ 
  then obtain i where  $mp' = mp \text{ } i$  and  $i: i \in \{0..n\}$  by auto
  with comp[unfolded simple-match-complete-wrt-def] cond[of i]
  have UNIQ-subst  $\sigma'$  (eq i) by auto
  from this[unfolded UNIQ-subst-alt-def] cond[of i] i
  have eq:  $\sigma' (x \text{ } i) = t \text{ } i \cdot \sigma'$  by auto
  from cond[of i] i have diff:  $\text{Var } (x \text{ } i) \neq t \text{ } i$  by auto
  from i have xi:  $x \text{ } i \in \text{Dom}$  unfolding Dom-def by auto
  with  $\delta$ [OF this] have  $\sigma' (x \text{ } i) \notin \text{Avoid}$  unfolding  $\sigma'$ -def by auto
  with eq have  $t \text{ } i \cdot \sigma' \notin \text{Avoid}$  by auto
  with  $\sigma'\sigma$  have  $t \text{ } i \notin \text{Tf}$  unfolding Avoid-def by auto
  from this[unfolded Tf-def] i obtain j where  $j: j \in \{0..n\}$  and  $ti: t \text{ } i = \text{Var}$ 
(x j)
  by auto
  from diff[unfolded ti] have diff:  $x \text{ } i \neq x \text{ } j$  by auto
  from eq[unfolded ti] have eq:  $\sigma' (x \text{ } i) = \sigma' (x \text{ } j)$  by auto
  from i j have inDom:  $\{x \text{ } i, x \text{ } j\} \subseteq \text{Dom}$  unfolding Dom-def by auto
  with eq have eq:  $\delta (x \text{ } i) = \delta (x \text{ } j)$  unfolding  $\sigma'$ -def by auto
  with inj inj' inDom diff
  have False
  by (metis (mono-tags, lifting) Diff-iff DomT-def Dom-def  $\langle t \text{ } i \notin \text{Tf} \rangle i$ 
image-eqI inj-onD
mem-Collect-eq ti)

```

```

    thus ?thesis ..
  qed
qed
end

end
theory FCF-Multiset
  imports
    FCF-Set
    Pattern-Completeness-Multiset
begin

fun depth-gterm :: ('f,'v)term  $\Rightarrow$  nat where
  depth-gterm (Fun f ts) = Suc (max-list (map depth-gterm ts))
| depth-gterm - = Suc 0

lemma depth-gterm-arg:  $t \in \text{set } ts \implies \text{depth-gterm } t < \text{depth-gterm } (\text{Fun } f \text{ } ts)$ 
  unfolding less-eq-Suc-le by (auto intro: max-list)

type-synonym ('f,'s)simple-match-problem-ms = ('f,nat  $\times$  's)term multiset multiset

type-synonym ('f,'s)simple-pat-problem-ms = ('f,'s)simple-match-problem-ms multiset

abbreviation mset2 :: ('f,'s)simple-match-problem-ms  $\Rightarrow$  ('f,'s)simple-match-problem
where
  mset2  $\equiv$  image set-mset o set-mset

abbreviation mset3 :: ('f,'s)simple-pat-problem-ms  $\Rightarrow$  ('f,'s)simple-pat-problem
where
  mset3  $\equiv$  image mset2 o set-mset

lemma mset2-simps:
  mset2 (add-mset eq mp) = insert (set-mset eq) (mset2 mp)
  set-mset (add-mset t eq) = insert t (set-mset eq)
  set-mset {#} = {}
  mset2 (mp1 + mp2) = mset2 mp1  $\cup$  mset2 mp2
  mset3 (add-mset mp p) = insert (mset2 mp) (mset3 p)
  by auto

context pattern-completeness-context
begin

```

inductive *smp-step-ms* :: ('f,'s)simple-match-problem-ms \Rightarrow ('f,'s)simple-match-problem-ms \Rightarrow bool

(**infix** $\langle \rightarrow_{ss} \rangle$ 50) **where**
smp-dup: $\text{add-mset } (\text{add-mset } t \ (\text{add-mset } t \ \text{eqc})) \ mp \ \rightarrow_{ss} \ \text{add-mset } (\text{add-mset } t \ \text{eqc}) \ mp$
| *smp-singleton*: $\text{add-mset } \{ \# \ t \ \# \} \ mp \ \rightarrow_{ss} \ mp$
| *smp-triv-sort*: $t : \iota \ \text{in } \mathcal{T}(C, \mathcal{V}) \ \Longrightarrow \ \text{cd-sort } \iota = 1 \ \Longrightarrow \ \text{add-mset } (\text{add-mset } t \ \text{eq}) \ mp \ \rightarrow_{ss} \ mp$
| *smp-decomp*: $(\bigwedge t. t \in \text{set-mset } \text{eqc} \ \Longrightarrow \ \text{root } t = \text{Some } (f, n))$
 $\Longrightarrow \ \text{eqcn} = \{ \# \{ \# \text{args } t ! \ i. t \in \# \ \text{eqc} \# \}. i \in \# \ \text{mset } [0..<n] \# \}$
 $\Longrightarrow (\bigwedge \text{eq}. \text{eq} \in \text{set-mset } \text{eqcn} \ \Longrightarrow \ \text{UNIQ } (\mathcal{T}(C, \mathcal{V}) \ ' \ \text{set-mset } \text{eq}))$
 $\Longrightarrow \ \text{add-mset } \text{eqc } mp \ \rightarrow_{ss} \ \text{eqcn} + mp$

inductive *smp-fail-ms* :: ('f,'s)simple-match-problem-ms \Rightarrow bool **where**

smp-clash: $\text{Conflict-Clash } s \ t \ \Longrightarrow$
 $s \in \text{eqc} \ \Longrightarrow \ t \in \text{eqc} \ \Longrightarrow \ \text{eqc} \in \text{mset2 } mp \ \Longrightarrow \ \text{smp-fail-ms } mp$
| *smp-decomp-fail*: $(\bigwedge t. t \in \text{set-mset } \text{eqc} \ \Longrightarrow \ \text{root } t = \text{Some } (f, n))$
 $\Longrightarrow \ i < n$
 $\Longrightarrow \neg \text{UNIQ } (\mathcal{T}(C, \mathcal{V}) \ ' \ (\lambda t. \text{args } t ! \ i) \ ' \ \text{set-mset } \text{eqc})$
 $\Longrightarrow \ \text{smp-fail-ms } (\text{add-mset } \text{eqc } mp)$

inductive *spp-step-ms* :: ('f,'s)simple-pat-problem-ms \Rightarrow ('f,'s)simple-pat-problem-ms *multiset* \Rightarrow bool

(**infix** $\langle \Rightarrow_{ss} \rangle$ 50) **where**
spp-solved: $\text{add-mset } \{ \# \} \ p \ \Rightarrow_{ss} \ \{ \# \}$
| *spp-simp*: $mp \ \rightarrow_{ss} \ mp' \ \Longrightarrow \ \text{add-mset } mp \ p \ \Rightarrow_{ss} \ \{ \# \text{add-mset } mp' \ p \# \}$
| *spp-delete*: $\text{smp-fail-ms } mp \ \Longrightarrow \ \text{add-mset } mp \ p \ \Rightarrow_{ss} \ \{ \# p \# \}$
| *spp-delete-large-sort*:
 $(\bigwedge i. i \leq (n :: \text{nat}) \ \Longrightarrow \ \text{snd } (x \ i) = \iota \wedge \text{eq } i \in \# \ mp \ i \wedge \text{Var } (x \ i) \neq t \ i \wedge \{ \text{Var } (x \ i), t \ i \} \subseteq \text{set-mset } (\text{eq } i)) \ \Longrightarrow$
 $x \ ' \ \{ 0..n \} \cap \text{tvars-spat } (\text{mset3 } p) = \{ \}$
 $(\text{card } (t \ ' \ \{ 0..n \}) < \text{card-of-sort } C \ \iota) \ \Longrightarrow$
 $p + \text{mset } (\text{map } mp \ [0..< \text{Suc } n]) \ \Rightarrow_{ss} \ \{ \# \ p \ \# \}$
| *spp-inst*: $\{ \# \{ \# \text{Var } x, t \# \} \# \} \in \# \ p$
 $\Longrightarrow \ \text{is-Fun } t$
 $\Longrightarrow \ \text{fst } ' \ \text{tvars-spat } (\text{mset3 } p) \cap \{ n..<n + m \} = \{ \}$
 $\Longrightarrow \ p \ \Rightarrow_{ss} \ \text{mset } (\text{map } (\lambda \tau. \text{image-mset } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau))))$
 $p) \ (\tau \text{s-list } n \ x)$
| *spp-split*: $mp = \text{add-mset } (\text{add-mset } s \ (\text{add-mset } t \ \text{eqc})) \ mp'$
 $\Longrightarrow \ \text{is-Var } s \neq \text{is-Var } t \ \Longrightarrow \ \text{eqc} \neq \{ \# \} \vee mp' \neq \{ \# \}$
 $\Longrightarrow \ \text{add-mset } mp \ p \ \Rightarrow_{ss} \ \{ \# \text{add-mset } \{ \# \{ \# \ s, t \# \} \# \} \ p, \ \text{add-mset } (\text{add-mset } (\text{add-mset } s \ \text{eqc}) \ mp') \ p \# \}$
end

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

begin

lemma *smp-fail-ms*: **assumes** *smp-fail-ms* *mp*
and *finite-constructor-form-pat* (*insert* (*mset2* *mp*) *p*)

```

shows finite-constructor-form-pat p
  simple-pat-complete C SS (insert (mset2 mp) p)  $\longleftrightarrow$  simple-pat-complete C SS p
proof –
  show finite-constructor-form-pat p using assms(2) unfolding finite-constructor-form-pat-def
by auto
  show simple-pat-complete C SS (insert (mset2 mp) p)  $\longleftrightarrow$  simple-pat-complete
  C SS p
  using assms
  proof (induct mp rule: smp-fail-ms.induct)
  case *: (smp-clash s t eqc mp)
  from *(4) obtain eq mp' where mp: mp = add-mset eq mp' and eqc: eqc =
  set-mset eq
  by (metis comp-apply image-iff mset-add)
  from eqc *(2–3) have set-mset eq = {s,t}  $\cup$  eqc by auto
  from eliminate-clash-spp(1)[OF *(5)[unfolded mset2-simps mp] refl refl this
  *(1)]
  show ?case unfolding mp mset2-simps .
  next
  case *: (smp-decomp-fail eqc f n i mp)
  from *(2–3) have ( $\forall$  eq  $\in$  {args t ! i |. t  $\in$  set-mset eqc} |. i  $\in$  {0..n}).
  UNIQ (T(C,V) ‘eq) = False
  by auto
  from decompose-spp(1)[OF *(4)[unfolded mset2-simps] refl refl *(1) refl refl,
  unfolded this if-False]
  show ?case unfolding mset2-simps by auto
  qed
qed

```

```

lemma smp-step-ms: assumes mp  $\rightarrow_{ss}$  mp'
  and finite-constructor-form-pat (insert (mset2 mp) p)
shows finite-constructor-form-pat (insert (mset2 mp') p)
  simple-pat-complete C SS (insert (mset2 mp) p)  $\longleftrightarrow$  simple-pat-complete C SS
  (insert (mset2 mp') p)
proof (atomize(full), insert assms, induct mp mp' rule: smp-step-ms.induct)
  case (smp-singleton t mp)
  from eliminate-uniq-spp[OF this[unfolded mset2-simps] refl refl refl]
  show ?case by (auto simp: Uniq-def)
next
  case *: (smp-triv-sort t  $\iota$  eq mp)
  show ?case
  proof (intro conjI)
  show finite-constructor-form-pat (insert (mset2 mp) p)
  using * unfolding finite-constructor-form-pat-def finite-constructor-form-mp-def
by auto
  show simple-pat-complete C SS (insert (mset2 (add-mset (add-mset t eq) mp))
  p) =
  simple-pat-complete C SS (insert (mset2 mp) p)
  unfolding simple-pat-complete-def bex-simps

```

```

proof (rule all-cong, intro disj-cong refl)
  fix  $\sigma$ 
  assume sig:  $\sigma :_s \mathcal{V} \mid 'SS \rightarrow \mathcal{T}(C)$ 
  from * obtain  $\tau$  where typed:  $u \in \text{insert } t \text{ (set-mset eq)} \implies u : \tau \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS)$  for  $u$ 
    unfolding finite-constructor-form-pat-def finite-constructor-form-mp-def by
  auto
  from typed-S-eq[OF this *(1)] have tau:  $\tau = \iota$  by auto
  from typed-imp-S[OF typed, of t] tau have  $\iota \in S$  by auto
  from cd[OF this] * k1
  have card: card-of-sort  $C \ \iota = 1$  by auto
  have UNIQ-subst  $\sigma$  (insert  $t$  (set-mset eq)) unfolding UNIQ-subst-alt-def
  proof (intro allI impI, goal-cases)
    case (1  $u \ v$ )
    {
      fix  $w$ 
      assume  $w \in \{u, v\}$ 
      with 1 typed tau have  $w : \iota \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS)$  by auto
      with sig have  $w \cdot \sigma : \iota \text{ in } \mathcal{T}(C)$  by (rule subst-hastype)
    }
  hence  $u \cdot \sigma : \iota \text{ in } \mathcal{T}(C) \ v \cdot \sigma : \iota \text{ in } \mathcal{T}(C)$  by auto
  with card
  show  $u \cdot \sigma = v \cdot \sigma$  unfolding card-of-sort-def card-eq-1-iff by auto
  qed
  thus simple-match-complete-wrt  $\sigma$  (mset2 (add-mset (add-mset  $t$  eq) mp)) =
    simple-match-complete-wrt  $\sigma$  (mset2 mp)
  unfolding simple-match-complete-wrt-def by simp
  qed
qed
next
  case *: (smp-decomp eqc  $f \ n \ eqcn \ mp$ )
  have eq: ( $\forall eq \in \text{mset2 } eqcn. \text{UNIQ } (\mathcal{T}(C, \mathcal{V}) \ 'eq) = \text{True}$ )
    using *(2-3) by auto
  define  $N$  where  $N = \{0..<n\}$ 
  have fin: finite  $N$  unfolding N-def by auto
  have mset2 eqcn =  $\{\{args \ t \ ! \ i \ \mid \ t \in \text{set-mset } eqc\} \ \mid \ i \in \{0..<n\}\}$ 
    unfolding *(2) mset-upt N-def[symmetric]
    apply (subst (2) finite-set-mset-mset-set[OF fin, symmetric])
  by (metis (no-types, lifting) comp-apply image-cong image-image multiset.set-map)
  from decompose-spp[OF *(4)[unfolded mset2-simps] refl refl *(1) this, unfolded
  eq if-True, OF - refl]
  show ?case unfolding mset2-simps by auto
  qed auto

lemma spp-step-ms-size: assumes  $p \Rightarrow_{s_s} Q$  and  $q \in \# \ Q$ 
  shows  $size \ q \leq size \ p$ 
  using assms by (cases, auto)

```

```

lemma spp-step-ms: assumes  $p \Rightarrow_{ss} Pn$ 
  and finite-constructor-form-pat (mset3  $p$ )
shows Ball (set-mset  $Pn$ ) ( $\lambda p'. \text{finite-constructor-form-pat } (mset3\ p')$ )
  simple-pat-complete  $C\ SS\ (mset3\ p) \longleftrightarrow \text{Ball } (set-mset\ Pn)\ (\lambda p'. \text{simple-pat-complete } C\ SS\ (mset3\ p'))$ 
proof (atomize(full), insert assms, induct  $p\ Pn$  rule: spp-step-ms.induct)
  case (spp-solved  $p$ )
    show ?case unfolding mset2-simps using detect-sat-spp by auto
  next
    case *: (spp-simp  $mp\ mp'\ p$ )
      from smp-step-ms[OF *(1) *(2)[unfolded mset2-simps]]
      show ?case unfolding mset2-simps by auto
  next
    case *: (spp-delete  $mp\ p$ )
      from smp-fail-ms[OF *(1) *(2)[unfolded mset2-simps]]
      show ?case unfolding mset2-simps by auto
  next
    case *: (spp-inst  $x\ t\ p\ n$ )
      from mset-add[OF *(1)] have mem:  $\{\{Var\ x,t\}\} \in mset3\ p$  by fastforce
      define  $p'$  where  $p' = mset3\ p$ 
      have id:  $\{(\cdot) ((\cdot) (\lambda t. t \cdot \tau)) ' mset3\ p \mid \tau \in \tau s\ n\ x\} = mset3\ ' \{image-mset\ (image-mset\ (image-mset\ (\lambda t. t \cdot \tau)))\ p \mid \tau \in \tau s\ n\ x\}$ 
      unfolding image-comp o-def image-mset.comp set-image-mset by auto
      from mem have  $x \in tvars\ spat\ (mset3\ p)$  unfolding  $p'$ -def[symmetric]
      by (auto intro!: beI[of -  $\{\{Var\ x, t\}\}$ ])
      from instantiate-spp[OF *(4,3) this refl]
      show ?case unfolding id using  $\tau s$ -list by auto
  next
    case *: (spp-split  $mp\ s\ t\ eqc\ mp'\ p$ )
      have insert  $s\ (insert\ t\ (set-mset\ eqc)) = \{s, t\} \cup set-mset\ eqc$  by auto
      from separate-var-fun-spp-single[OF *(4)[unfolded mset2-simps *(1)] refl refl this refl]
      show ?case unfolding mset2-simps *(1) by auto
  next
    case *: (spp-delete-large-sort  $n\ x\ \iota\ eq\ mp\ t\ p$ )
      have  $mset3\ (p + mset\ (map\ mp\ [0..<Suc\ n])) = mset3\ p \cup (mset2\ o\ mp) ' set\ [0..<Suc\ n]$ 
      by auto
      also have  $set\ [0..<Suc\ n] = \{0..n\}$  by auto
      finally have id:  $mset3\ (p + mset\ (map\ mp\ [0..<Suc\ n])) = mset3\ p \cup (mset2\ o\ mp) ' \{0..n\}$  by auto
      have main: simple-pat-complete  $C\ SS\ (mset3\ (p + mset\ (map\ mp\ [0..<Suc\ n])))$ 
      =
      simple-pat-complete  $C\ SS\ (mset3\ p)$  unfolding id
      apply (rule eliminate-large-sort[OF - *(2,3), of set-mset o eq])
      subgoal for  $i$  using *(1)[of  $i$ ] by auto
      subgoal using *(4) unfolding id by (auto simp: finite-constructor-form-pat-def)
      done
      show ?case unfolding main using *(4) by (auto simp: finite-constructor-form-pat-def)

```

qed

lemma *finite-tvars-spat-mset3*: *finite (tvars-spat (mset3 p))*
by (*intro finite-Union; fastforce*)

lemma *finite-tvars-smp-mset2*: *finite (tvars-smp (mset2 mp))*
by (*intro finite-Union; fastforce*)

lemma *normal-form-spp-step-fvf*: **assumes** *finite-constructor-form-pat (mset3 p)*

and $\nexists P. p \Rightarrow_{ss} P \wedge P \neq \{\#\}$
and *mp*: *mp* \in *mset3 p*
and *eqc*: *eqc* \in *mp*
and *t*: *t* \in *eqc*
shows *is-Var t*
proof (*rule ccontr*)
 assume *f*: *is-Fun t*
 from *mp* **obtain** *mp'* **where** *mp'*: *mp'* \in *set-mset p* **and** *mp*: *mp* = *mset2 mp'*
by *auto*
 from *eqc*[*unfolded mp*] **obtain** *eqc'* **where** *eqc'*: *eqc'* \in *set-mset mp'* **and** *eqc*: *eqc*
= *set-mset eqc'* **by** *auto*
 from *t*[*unfolded eqc*] **obtain** *eqc2* **where** *id1*: *eqc'* = *add-mset t eqc2* **by** (*rule*
mset-add)
 from *eqc'* **obtain** *mp2* **where** *id2*: *mp'* = *add-mset eqc' mp2* **by** (*rule mset-add*)
 from *mp'* **obtain** *p2* **where** *id3*: *p* = *add-mset mp' p2* **by** (*rule mset-add*)
 have *p*: *p* = *add-mset (add-mset (add-mset t eqc2) mp2) p2* **unfolding** *id1 id2*
id3 ..
 with *assms* **have** *contra*: *add-mset (add-mset (add-mset t eqc2) mp2) p2* \Rightarrow_{ss}
P \Rightarrow *P* $\neq \{\#\}$ \Rightarrow *False* **for** *P* **by** *auto*
 from *assms*(1)[*unfolded finite-constructor-form-pat-def p*]
 have *fin*: *finite-constructor-form-mp (mset2 (add-mset (add-mset t eqc2) mp2))*
by *auto*
 show *False*
 proof (*cases eqc2 = \{\#\}*)
 case *True*
 show *False*
 apply (*rule contra*)
 apply (*unfold True*)
 by (*rule spp-simp, rule smp-singleton, auto*)
 next
 case *ne*: *False*
 show *False*
 proof (*cases* $\exists s \in$ *set-mset eqc2*. *is-Var s*)
 case *True*
 then **obtain** *s eqc3* **where** *eqc2* = *add-mset s eqc3* **and** *is-Var s* **by** (*metis*
mset-add)
 then **obtain** *x* **where** *eqc*: *add-mset t eqc2* = *add-mset (Var x) (add-mset t*
eqc3) **by** (*cases s, auto*)
 from *fin*[*unfolded eqc finite-constructor-form-mp-def*] **obtain** ι

```

    where  $Var\ x : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by auto
  hence  $xS: snd\ x \in S$  unfolding hastype-def by (auto simp: restrict-map-def
split: if-splits)
  show False
  proof (cases  $eqc3 = \{\#\} \wedge mp2 = \{\#\}$ )
    case True
      hence True:  $eqc3 = \{\#\} \wedge mp2 = \{\#\}$  by auto
      define names where  $names = fst\ ' \cup\ (\cup\ (\cup\ ((')\ ((')\ vars)\ ' \text{insert}\ (\text{insert}\ \{Var\ x, t\}\ (mset2\ \{\#\}))\ (mset3\ p2))))$ 
      define n where  $n = Max\ names$ 
      have id:  $mset3\ (\text{add-mset}\ \{\#\{\#\ Var\ x, t\}\#\}\ p2) =$ 
         $\text{insert}\ (\text{insert}\ \{Var\ x, t\}\ (mset2\ \{\#\}))\ (mset3\ p2)$  by auto
      have  $names = fst\ ' \text{tvars-spat}\ (mset3\ p)$  unfolding p eqc True names-def
by simp
      also have finite ... using finite-tvars-spat-mset3[of p] by blast
      finally have  $names: \bigwedge k. k \in names \implies k \leq n$  unfolding n-def by auto
      from not-empty-sort[OF xS, unfolded empty-sort-def]
      obtain t where  $t : snd\ x$  in  $\mathcal{T}(C)$  by auto
      then obtain f ss where  $f : ss \rightarrow snd\ x$  in  $C$ 
        by (induct, auto)
      hence  $set\ (Cl\ (snd\ x)) \neq \{\}$  unfolding Cl by auto
      hence  $Cl: Cl\ (snd\ x) \neq []$  by auto
      show False
        apply (rule contra)
        apply (unfold eqc, unfold True)
        apply (rule spp-inst[OF - f, of x - Suc n])
        apply force
        apply (unfold id)
        apply (unfold names-def[symmetric])
        apply (insert names, fastforce)
        apply (insert Cl, auto simp: \tau s-list-def)
        done
    next
      case False
      hence diff:  $eqc3 \neq \{\#\} \vee mp2 \neq \{\#\}$  by auto
      from contra[OF spp-split[OF - - diff], unfolded eqc True, OF refl] f
      show ?thesis by auto
  qed
next
case False
  hence funs:  $\bigwedge s. s \in \# eqc2 \implies is-Fun\ s$  by auto
  show False
  proof (cases  $\exists s \in \# eqc2. root\ s \neq root\ t$ )
    case True
      then obtain s eqc3 where  $eqc2: eqc2 = \text{add-mset}\ s\ eqc3$  and diff:  $root\ s$ 
 $\neq root\ t$ 
        by (meson mset-add)
      from funs[of s] eqc2 diff f obtain f g where  $rt: root\ t = Some\ f\ root\ s =$ 
Some g and diff:  $f \neq g$ 

```

```

    by (cases s; cases t; auto)
  hence conf: Conflict-Clash s t by (cases s; cases t; auto simp: conflicts.simps)
  show False
    apply (rule contra)
    apply (unfold eqc2)
    apply (rule spp-delete)
    apply (rule smp-clash[OF conf, of insert t (insert s (set-mset eqc3))])
    by auto
  next
  case False
  from f obtain f n where root t = Some (f,n) by (cases t, auto)
  with False have rt:  $\bigwedge s. s \in \# \text{ add-mset } t \text{ eqc2} \implies \text{root } s = \text{Some } (f,n)$ 
by auto
  let ?cond =  $\forall eq \in \# \{ \# \{ \# \text{args } t ! i. t \in \# \text{ add-mset } t \text{ eqc2} \# \}. i \in \# \text{ mset } [0..<n] \# \}. \text{UNIQ } (\mathcal{T}(C,\mathcal{V}) \text{ ' set-mset } eq)$ 
  show False
  proof (cases ?cond)
  case True
  show False
    apply (rule contra)
    apply (rule spp-simp)
    apply (rule smp-decomp[OF rt refl])
    using True by auto
  next
  case False
  define eqc3 where eqc3 = add-mset t eqc2
  from False obtain i where i:  $i < n$  and nuniq:  $\neg \text{UNIQ } (\mathcal{T}(C,\mathcal{V}) \text{ ' set-mset } \{ \# \text{args } t ! i. t \in \# \text{ add-mset } t \text{ eqc2} \# \})$ 
  unfolding mset2-simps by auto
  show False
    apply (rule contra)
    apply (rule spp-delete)
    apply (rule smp-decomp-fail[OF rt i])
    using nuniq by auto
  qed
  qed
  qed
  qed
  qed

```

Every normal form consists purely of variables, and these variable are of sorts that have a small cardinality (upper bound is the number of matching problems in p).

lemma *NF-spp-step-fvf-with-small-card*: **assumes** *finite-constructor-form-pat* (*mset3 p*)
and $\nexists P. p \Rightarrow_{ss} P \wedge P \neq \{ \# \}$
and *mp*: $mp \in \# p$
and *eqc*: $eqc \in \# mp$
and *t*: $t \in \# eqc$

shows $\exists x \iota. t = \text{Var } (x, \iota) \wedge \text{card-of-sort } C \iota \leq \text{size } p$
proof –
from *normal-form-spp-step-fvf*[*OF* *assms*(1–2), *of* *mset2* *mp* *set-mset* *eqc* *t*] *mp*
eqc *t*
obtain $x \iota$ **where** $tx: t = \text{Var } (x, \iota)$ **by** *force*
from *assms*(1)[*unfolded* *finite-constructor-form-pat-def* *finite-constructor-form-mp-def*,

rule-format, *of* *mset2* *mp* *set-mset* *eqc*] *mp* *eqc* *t* *tx*
have $ts: t : \iota$ **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$
by *auto* (*metis* *Term.simps*(1) *comp-eq-dest-lhs* *hastype-def* *snd-eqD* *typed-S-eq*)
with tx **have** $iota: \iota \in S$ **by** (*metis* *typed-imp-S*)
show *?thesis*
proof (*intro* *exI* *conjI*, *rule* tx)
show *card-of-sort* $C \iota \leq \text{size } p$
proof (*rule* *ccontr*)
assume $\neg ?thesis$
hence *large*: *card-of-sort* $C \iota > \text{size } p$ **by** *auto*
define *filt* **where** $\text{filt } mp = (\exists \text{eqc} \in \# \text{mp}. \exists t \in \# \text{eqc}. t : \iota \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS))$
for *mp*
define *p1* **where** $p1 = \text{filter-mset } \text{filt } p$
define *p2* **where** $p2 = \text{filter-mset } (\text{Not } o \text{filt}) p$
have $p: p = p1 + p2$ **unfolding** *p1-def* *p2-def* **by** *auto*
have *large*: *card-of-sort* $C \iota > \text{size } p1$ **using** *large* **unfolding** *p* **by** *auto*
have $mp: mp \in \# p1$ **unfolding** *p1-def* *filt-def* **using** *mp* *eqc* *t* *ts* **by** *auto*
hence $p1: p1 \neq \{\#\}$ **by** *auto*
obtain *p1l* **where** $p1l: p1 = \text{mset } p1l$ **by** (*metis* *ex-mset*)
have $len: \text{length } p1l = \text{size } p1$ **unfolding** *p1l* **by** *auto*
with *p1* **obtain** n **where** $lenp: \text{length } p1l = \text{Suc } n$ **by** (*cases* *p1l*, *auto*)
define *mp* **where** $mp \ i = p1l \ ! \ i$ **for** i
{
fix i
assume $i \leq n$
hence $i < \text{length } p1l$ **unfolding** *lenp* **by** *auto*
hence $mp \ i \in \text{set } p1l$ **unfolding** *mp-def* **by** *auto*
hence $mp: mp \ i \in \# p1$ **unfolding** *p1l* **by** *auto*
from *this*[*unfolded* *p1-def*] **have** $mpp: mp \ i \in \# p$ **and** *filt*: $\text{filt } (mp \ i)$ **by**
auto
from *assms*(1)[*unfolded* *finite-constructor-form-pat-def*] *mpp*
have *fin*: *finite-constructor-form-mp* (*mset2* (*mp* i)) **by** *auto*
from *filt*[*unfolded* *filt-def*] **obtain** *eqc* t **where** $\text{eqc} \in \# mp \ i$
and $t: t \in \# \text{eqc}$ **and** $ts: t : \iota$ **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$
by *auto*
from *fin*[*unfolded* *finite-constructor-form-mp-def*, *rule-format*, *of* *set-mset*
eqc] *eqc*
obtain ι' **where** $\text{eqcs}: \bigwedge s. s \in \# \text{eqc} \implies s : \iota'$ **in** $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ **by** *auto*
from eqcs [*OF* t] *ts* **have** $\iota' = \iota$ **by** *fastforce*
note $\text{eqcs} = \text{eqcs}$ [*unfolded* *this*]
{
fix mp'

```

assume  $mp\ i \rightarrow_{ss} mp'$ 
from  $spp\text{-}simp[OF\ this,\ of\ p - \{\#\ mp\ i\ \#\}]$   $mpp$ 
have  $\exists P. p \Rightarrow_{ss} P \wedge P \neq \{\#\}$  by  $auto$ 
with  $assms$  have  $False$  by  $auto$ 
} note  $nf = this$ 
from  $eqc$  obtain  $mp'$  where  $mpi: mp\ i = add\text{-}mset\ eqc\ mp'$  by ( $metis\ mset\text{-}add$ )
from  $t$  obtain  $eqc'$  where  $eqct: eqc = add\text{-}mset\ t\ eqc'$  by ( $metis\ mset\text{-}add$ )
from  $smp\text{-}singleton[of\ t\ mp']$   $nf[unfolding\ mpi\ eqct]$  have  $eqc' \neq \{\#\}$  by
 $auto$ 
then obtain  $s\ eqc2$  where  $eqc' = add\text{-}mset\ s\ eqc2$  by ( $cases\ eqc',\ auto$ )
note  $eqct = eqct[unfolding\ this]$ 
from  $smp\text{-}dup[of\ t\ eqc2\ mp']$   $nf[unfolding\ mpi\ eqct]$ 
have  $st: s \neq t$  by  $auto$ 
from  $eqct$  have  $s: s \in\#\ eqc$  by  $auto$ 
from  $normal\text{-}form\text{-}spp\text{-}step\text{-}fvf[OF\ assms(1-2),\ of\ mset2\ (mp\ i)\ set\text{-}mset\ eqc,\ OF\ -\ -\ this]$ 
 $mpp\ eqc$  obtain  $x$  where  $sx: s = Var\ x$  by  $auto$ 
from  $eqcs[OF\ s,\ unfolding\ sx]$  have  $snd: snd\ x = \iota$ 
by ( $metis\ Term.\text{simps}(1)\ comp\text{-}apply\ hastypeD\ option.\text{inject}\ typed\text{-}restrict\text{-}imp\text{-}typed$ )
have  $summary:$ 
 $snd\ x = \iota \wedge eqc \in\#\ mp\ i \wedge Var\ x \neq t \wedge \{Var\ x,\ t\} \subseteq set\text{-}mset\ eqc \wedge t : \iota$ 
in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ 
using  $st\ snd\ eqc\ eqct\ sx\ ts$  by  $auto$ 
hence  $\exists x\ eqc\ t. snd\ x = \iota \wedge eqc \in\#\ mp\ i \wedge Var\ x \neq t \wedge \{Var\ x,\ t\} \subseteq$ 
 $set\text{-}mset\ eqc \wedge t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by  $blast$ 
}
hence  $\forall i. \exists x\ eqc\ t. i \leq n \longrightarrow snd\ x = \iota \wedge eqc \in\#\ mp\ i \wedge Var\ x \neq t \wedge$ 
 $\{Var\ x,\ t\} \subseteq set\text{-}mset\ eqc \wedge t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by  $blast$ 
from  $choice[OF\ this]$  obtain  $x$  where
 $\forall i. \exists eqc\ t. i \leq n \longrightarrow snd\ (x\ i) = \iota \wedge eqc \in\#\ mp\ i \wedge Var\ (x\ i) \neq t \wedge$ 
 $\{Var\ (x\ i),\ t\} \subseteq set\text{-}mset\ eqc \wedge t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by  $blast$ 
from  $choice[OF\ this]$  obtain  $eqc$  where
 $\forall i. \exists t. i \leq n \longrightarrow snd\ (x\ i) = \iota \wedge eqc\ i \in\#\ mp\ i \wedge Var\ (x\ i) \neq t \wedge \{Var$ 
 $(x\ i),\ t\} \subseteq set\text{-}mset\ (eqc\ i) \wedge t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by  $blast$ 
from  $choice[OF\ this]$  obtain  $t$  where
 $witnesses: \bigwedge i. i \leq n \implies snd\ (x\ i) = \iota \wedge eqc\ i \in\#\ mp\ i \wedge Var\ (x\ i) \neq t \wedge$ 
 $\{Var\ (x\ i),\ t\} \subseteq set\text{-}mset\ (eqc\ i)$ 
and  $ti: \bigwedge i. i \leq n \implies t\ i : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by  $blast$ 
note  $step = spp\text{-}delete\text{-}large\text{-}sort[of\ n\ x\ \iota\ eqc\ mp\ t\ p2,\ OF\ witnesses]$ 
have  $card\ (t\ ' \{0..n\}) \leq card\ \{0..n\}$  using  $card\text{-}image\text{-}le$  by  $blast$ 
also have  $\dots = size\ p1$  using  $len\ lenp\ p1l$  by  $auto$ 
also have  $\dots < card\text{-}of\text{-}sort\ C\ \iota$  by  $fact$ 
finally have  $card\ (t\ ' \{0..n\}) < card\text{-}of\text{-}sort\ C\ \iota$  .
note  $step = step[OF\ -\ -\ this]$ 
have  $id: map\ mp\ [0..<Suc\ n] = p1l$  unfolding  $mp\text{-}def$  using  $lenp$  by ( $rule\ map\text{-}nth'$ )
have  $p2 + mset\ (map\ mp\ [0..<Suc\ n]) = p$  unfolding  $id\ p\ p1l$  by  $auto$ 
note  $step = step[unfolding\ this]$ 

```

```

let ?Var = Var :: nat × 's ⇒ ('f, nat × 's) Term.term
define Vs where Vs = tvars-spat (mset3 p2)
have x ' {0..n} ∩ tvars-spat (mset3 p2) = {}
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain s where s1: s ∈ x ' {0..n}
    and s2: s ∈ tvars-spat (mset3 p2) unfolding Vs-def[symmetric] by blast
  from s1 obtain i where i ≤ n and s: s = x i by auto
  with witnesses[of i] ti[of i] have si: Var s : ι in T(C, V)
    by (metis Term.simps(1) comp-eq-dest-lhs hastypeI)
  from s2 obtain mp eqc u where *: mp ∈# p2 eqc ∈# mp u ∈# eqc s ∈
vars u
    by fastforce
  from *(1)[unfolded p2-def] have mp: mp ∈# p and ¬ filt mp by auto
  from this(2)[unfolded filt-def] * have ns: ¬ u : ι in T(C, V | SS) by auto
  from normal-form-spp-step-fvf[OF assms(1-2), of mset2 mp set-mset eqc
u] mp *
    have is-Var u by auto
  with * have u: u = Var s by auto
  note ns = ns[unfolded this]
  from si iota u ns show False
    by (smt (verit, del-Insts) SigmaE SigmaI UNIV-I UNIV-Times-UNIV
Var-hastype comp-apply eq-Some-iff-hastype
hastype-restrict option.inject snd-conv)
  qed
  note step = step[OF - this]
  from step assms(2) show False by auto
  qed
  qed
  qed

```

definition *max-depth-sort* :: 's ⇒ nat **where**
max-depth-sort s = Maximum (depth-gterm ' {t. t : s in T(C)})

lemma *max-depth-sort-wit*: **assumes** *finite-sort* C s
and s ∈ S

shows ∃ t. t : s in T(C) ∧
depth-gterm t = max-depth-sort s ∧
(∀ t'. t' : s in T(C) → depth-gterm t' ≤ depth-gterm t)

proof –

```

let ?T = {t. t : s in T(C)}
from sorts-non-empty[OF assms(2)] have depth-gterm ' ?T ≠ {} by auto
moreover from assms(1)[unfolded finite-sort-def] have fin: finite (depth-gterm
' ?T) by auto
ultimately obtain d where d ∈ depth-gterm ' ?T and d: d = Maximum
(depth-gterm ' ?T)
by (metis has-MaximumD(1) has-Maximum-nat-iff-finite)
from this obtain t where t: t ∈ ?T

```

and *depth*: *depth-gterm* $t = \text{Maximum} (\text{depth-gterm } \text{' ?T}) d = \text{depth-gterm } t$
by *auto*
have $t' \in \text{' ?T} \implies \text{depth-gterm } t' \leq \text{depth-gterm } t$ **for** t'
unfolding *depth*(' ?)[*symmetric*] **unfolding** d **using** *fin*
by (*meson* *bdd-above-Maximum-nat* *bdd-above-nat* *image-iff*)
thus *?thesis* **unfolding** *max-depth-sort-def* *depth*(1)[*symmetric*] **using** t
by (*intro* *exI*[*of - t*], *auto*)
qed

lemma *max-depth-sort-Fun*: **assumes** $f : \sigma s \rightarrow s$ *in* C
and si : $si \in \text{set } \sigma s$
and $fins$: *finite-sort* C s
shows *max-depth-sort* $si < \text{max-depth-sort } s$
proof –
from *C-sub-S*[*OF* f] si **have** $s : s \in S$ **and** siS : $si \in S$ **by** *auto*
from *finite-arg-sort*[*OF* $fins$ f si]
have *finite-sort* C si .
from *max-depth-sort-wit*[*OF* *this* siS] **obtain** ti **where**
 ti : $ti : si \text{ in } \mathcal{T}(C)$ **and** $depi$: *depth-gterm* $ti = \text{max-depth-sort } si$ **by** *auto*
define t **where** $t s = (\text{SOME } t. t : s \text{ in } \mathcal{T}(C))$ **for** s
have t : $s \in \text{set } \sigma s \implies t s : s \text{ in } \mathcal{T}(C)$ **for** s
using *someI-ex*[*OF* *sorts-non-empty*[*of* s]] *C-sub-S*[*OF* f]
unfolding *t-def* **by** *auto*
from si **obtain** i **where** si : $si = \sigma s ! i$ **and** i : $i < \text{length } \sigma s$ **by** (*auto* *simp*:
set-conv-nth)
define trm **where** $trm = \text{Fun } f (\text{map } (\lambda j. \text{if } j = i \text{ then } ti \text{ else } t (\sigma s ! j))$
 $[0..<\text{length } \sigma s])$
have trm : $trm : s \text{ in } \mathcal{T}(C)$
unfolding *trm-def*
apply (*intro* *Fun-hastypeI*[*OF* f] *list-all2-all-nthI*)
apply *force*
subgoal **for** j **using** t si ti **by** (*auto* *simp*: *set-conv-nth*)
done
have *max-depth-sort* $si = \text{depth-gterm } ti$ **using** $depi$ **by** *auto*
also **have** $\dots < \text{depth-gterm } trm$ **unfolding** *trm-def*
by (*rule* *depth-gterm-arg*, *insert* i , *auto*)
also **have** $\dots \leq \text{max-depth-sort } s$
using *max-depth-sort-wit*[*OF* $fins$ s] trm **by** *auto*
finally **show** *?thesis* .
qed

lemma *max-depth-sort-var*: **assumes** $t : s \text{ in } \mathcal{T}(C, \mathcal{V} \mid \text{' SS})$
and $x \in \text{vars } t$
and *finite-sort* C s
shows *max-depth-sort* (*snd* x) $\leq \text{max-depth-sort } s$
using *assms*
proof (*induct*)
case (*Var* y s)
thus *?case* **by** (*auto* *simp*: *hastype-def* *restrict-map-def* *split*: *if-splits*)

next
case (*Fun* *f* *ts* σ *s* τ)
from *Fun*(4) **obtain** *i* **where** $i < \text{length } ts$ **and** $x \in \text{vars } (ts ! i)$
by (*auto simp: set-conv-nth*)
from *i* *Fun*(2) **have** $\text{mem}: \sigma s ! i \in \text{set } \sigma s$ **by** (*auto simp: set-conv-nth list-all2-conv-all-nth*)
from *finite-arg-sort*[*OF Fun*(5,1) *mem*]
have *finite-sort* *C* ($\sigma s ! i$) .
with *Fun*(3) *i* *x* **have** $\text{max-depth-sort } (\text{snd } x) \leq \text{max-depth-sort } (\sigma s ! i)$
by (*auto simp: list-all2-conv-all-nth*)
with *max-depth-sort-Fun*[*OF Fun*(1) *mem Fun*(5)]
show ?*case* **by** *simp*
qed

definition *max-depth-sort-smp* :: (*f, 's*) *simple-match-problem-ms* \Rightarrow *nat* **where**
max-depth-sort-smp *mp* = *Max* (*insert* 0 (*max-depth-sort 'snd 'tvars-smp* (*mset2*
mp)))

definition *max-depth-sort-p* :: (*f, 's*) *simple-pat-problem-ms* \Rightarrow *nat* **where**
max-depth-sort-p *p* = *sum-mset* (*image-mset* *max-depth-sort-smp* *p*)

lemma *max-depth-sort-p-add*[*simp*]: *max-depth-sort-p* (*add-mset* *mp* *p*) =
max-depth-sort-smp *mp* + *max-depth-sort-p* *p*
unfolding *max-depth-sort-p-def* **by** *simp*

lemma *finite-constructor-form-pat-add*: *finite-constructor-form-pat* (*mset3* (*add-mset*
mp *p*))
= (*finite-constructor-form-mp* (*mset2* *mp*) \wedge *finite-constructor-form-pat* (*mset3*
p))
unfolding *finite-constructor-form-pat-def* **by** *auto*

lemma *mds-decrease-inst*: **assumes** *ft: is-Fun* *t*
and *tau*: $\tau \in \tau s$ *n* *x*
and *mp*: $mp = \{\#\{\# \text{Var } x, t\#\}\#\}$
and *fin*: *finite-constructor-form-mp* (*mset2* *mp*)
shows *max-depth-sort-smp* *mp* > *max-depth-sort-smp* (*image-mset* (*image-mset*
 $(\lambda t. t \cdot \tau)$) *mp*)
proof –
from *fin*[*unfolded finite-constructor-form-mp-def* *mp*] **obtain** ι
where *fin*: *finite-sort* *C* ι **and** $x: \text{Var } x : \iota$ *in* $\mathcal{T}(C, \mathcal{V} \mid 'SS)$ **and** $t: t : \iota$ *in*
 $\mathcal{T}(C, \mathcal{V} \mid 'SS)$
by *auto*
from *x* **have** *iota*: $\iota = \text{snd } x$ **and** $xS: \text{snd } x \in S$ **by** (*auto simp: hastype-def*
restrict-map-def split: if-splits)
from *ft* **obtain** *f* *ts* **where** *ft*: $t = \text{Fun } f \text{ } ts$ **by** (*cases* *t*, *auto*)
from *t*[*unfolded ft Fun-hastype*] **obtain** σs **where** $f: f : \sigma s \rightarrow \iota$ *in* *C*
and $ts: ts :_{\iota} \sigma s$ *in* $\mathcal{T}(C, \mathcal{V} \mid 'SS)$
by *auto*
{
fix *y*

```

assume  $y \in \text{vars } t$ 
from  $\text{this}[\text{unfolded } ft]$  obtain  $ti$  where  $y: y \in \text{vars } ti$  and  $ti: ti \in \text{set } ts$  by
auto
from  $ti$   $ts$  obtain  $\sigma i$  where  $\sigma i: \sigma i \in \text{set } \sigma s$  and  $ti: ti : \sigma i$  in  $\mathcal{T}(C, \mathcal{V} \mid \text{' } SS)$ 
  unfolding  $\text{list-all2-conv-all-nth set-conv-nth}$  by force
from  $\text{max-depth-sort-var}[OF \text{ } ti \text{ } y \text{ } \text{finite-arg-sort}[OF \text{ } fin \text{ } f \text{ } \sigma i]]$ 
have  $\text{max-depth-sort } (snd \text{ } y) \leq \text{max-depth-sort } \sigma i$  .
also have  $\dots < \text{max-depth-sort } (snd \text{ } x)$ 
  using  $\text{max-depth-sort-Fun}[OF \text{ } f \text{ } \sigma i \text{ } fin]$  iota by auto
finally have  $\text{max-depth-sort } (snd \text{ } y) < \text{max-depth-sort } (snd \text{ } x)$  by auto
} note  $\text{max-x} = \text{this}$ 

from  $\text{max-depth-sort-wit}[OF \text{ } fin, \text{ unfolded } \text{iota}, OF \text{ } xS]$  obtain  $wx$ 
where  $wx: wx : snd \text{ } x$  in  $\mathcal{T}(C)$  and  $\text{to-wx}: \text{max-depth-sort } (snd \text{ } x) = \text{depth-gterm}$ 
wx
and  $wx\text{-max}: \bigwedge t'. t' : snd \text{ } x$  in  $\mathcal{T}(C) \implies \text{depth-gterm } t' \leq \text{depth-gterm } wx$ 
by auto

have  $\text{max-depth-sort-smp } mp = \text{Max } (\text{max-depth-sort } \text{' } snd \text{' } \text{insert } x \text{ } (\text{vars } t))$ 
unfolding  $mp$ 
   $\text{max-depth-sort-smp-def}$  by auto
also have  $\dots = \text{max-depth-sort } (snd \text{ } x)$ 
  by (rule Max-eqI, insert max-x, force+)
finally have  $\text{max-mp}: \text{max-depth-sort-smp } mp = \text{max-depth-sort } (snd \text{ } x)$  .

from  $\text{tau}[\text{unfolded } \tau s\text{-def}]$  obtain  $f \text{ } \sigma s$  where  $f: f : \sigma s \rightarrow snd \text{ } x$  in  $C$ 
and  $\text{tau}: \tau = \tau c \text{ } n \text{ } x \text{ } (f, \sigma s)$  by auto
have  $\text{max-depth-sort-smp } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau)) \text{ } mp) =$ 
   $\text{Max } (\text{insert } 0 \text{ } (\text{max-depth-sort } \text{' } snd \text{' } (\text{vars } (\tau \text{ } x) \cup \text{vars } (t \cdot \tau))))$ 
unfolding  $\text{max-depth-sort-smp-def } mp$  by simp
also have  $\dots \leq \text{Max } (\text{insert } 0 \text{ } (\text{max-depth-sort } \text{' } (snd \text{' } \text{vars } (\tau \text{ } x) \cup \text{snd \text{' } vars}$ 
t)))
by (rule Max-mono, auto simp: vars-term-subst tau tau-c-def subst-def)
also have  $\text{snd \text{' } vars } (\tau \text{ } x) = \text{set } \sigma s$ 
by (simp add: tau tau-c-def subst-def) (simp flip: set-map)
also have  $\text{Max } (\text{insert } 0 \text{ } (\text{max-depth-sort } \text{' } (\text{set } \sigma s \cup \text{snd \text{' } vars } t))) < \text{max-depth-sort}$ 
(snd x)
proof (subst Max-less-iff, force, force, intro ballI)
  fix  $d$ 
assume  $d: d \in \text{insert } 0 \text{ } (\text{max-depth-sort } \text{' } (\text{set } \sigma s \cup \text{snd \text{' } vars } t))$ 
show  $d < \text{max-depth-sort } (snd \text{ } x)$ 
proof (cases d \in \text{max-depth-sort \text{' } snd \text{' } vars } t)
  case True
    with  $\text{max-x}$  show ?thesis by auto
  next
    case False
hence  $d = 0 \vee d \in \text{max-depth-sort } \text{' } \text{set } \sigma s$  using  $d$  by auto
thus ?thesis
proof

```

```

    assume d = 0
    thus ?thesis unfolding to-wx by (cases wx, auto)
  next
    assume d ∈ max-depth-sort ‘ set σ s
    then obtain σ where mem: σ ∈ set σ s and d: d = max-depth-sort σ by
auto
    from max-depth-sort-Fun[OF f mem] d fin iota show ?thesis by auto
  qed
  qed
  qed
  finally show ?thesis unfolding max-mp .
qed

```

```

lemma mds-weak-decrease-inst: assumes tau: τ ∈ τ s n x
  and fin: finite-sort C (snd x)
shows max-depth-sort-smp mp ≥ max-depth-sort-smp (image-mset (image-mset
(λ t. t · τ)) mp)
proof -
  from tau[unfolded τ s-def] obtain f σ s where f: f : σ s → snd x in C
  and tau: τ = τ c n x (f, σ s) by auto
  note tau = tau[unfolded τ c-def split]
  show ?thesis
  proof (cases x ∈ tvars-smp (mset2 mp))
    case False
    have image-mset (image-mset (λ t. t · τ)) mp = image-mset (image-mset (λ t.
t · Var)) mp
    proof (intro image-mset-cong refl term-subst-eq, goal-cases)
      case (1 eqc t y)
      with False have x ≠ y by auto
      thus ?case unfolding tau by (auto simp: subst-def)
    qed
    also have ... = mp by simp
    finally show ?thesis by simp
  next
    case True
    define tv where tv = tvars-smp (mset2 mp)
    let ?mp = image-mset (image-mset (λ t. t · τ)) mp
    from True have x: x ∈ tv unfolding tv-def .
    have fin: finite tv unfolding tv-def by auto
    have max-depth-sort-smp mp = Max (insert 0 (max-depth-sort ‘ snd ‘ (insert
x tv)))
    unfolding max-depth-sort-smp-def tv-def using True by auto
    also have ... = Max (max-depth-sort ‘ snd ‘ (insert x tv)) using fin by simp
    finally have max-mp: max-depth-sort-smp mp = Max (max-depth-sort ‘ snd ‘
insert x tv) .

```

```

  have tvars-smp (mset2 ?mp) = ⋃ (vars ‘ τ ‘ tv)
  by (fastforce simp add: tv-def vars-term-subst)
  also have ... = vars (τ x) ∪ ⋃ (vars ‘ τ ‘ tv) using x by auto

```

also have $\dots = \text{vars } (\tau x) \cup (tv - \{x\})$ **unfolding** *tau subst-def* **by** *auto*
finally have $\text{tvars: tvars-smp } (mset2 \text{ ?mp}) = \text{vars } (\tau x) \cup (tv - \{x\})$.

have $\text{max-depth-sort-smp } ?mp = \text{Max } (\text{insert } 0 \text{ } (\text{max-depth-sort } ' (\text{snd } ' \text{vars } (\tau x) \cup \text{snd } ' (tv - \{x\}))))$
unfolding *max-depth-sort-smp-def* **tvars by** (*metis image-Un*)
also have $\text{snd } ' \text{vars } (\tau x) = \text{set } \sigma s$
by (*simp add: tau subst-def*) (*simp flip: set-map*)
also have $\text{Max } (\text{insert } 0 \text{ } (\text{max-depth-sort } ' (\text{set } \sigma s \cup \text{snd } ' (tv - \{x\}))))$
 $\leq \text{max-depth-sort-smp } mp$ **unfolding** *max-mp*
proof (*intro Max-le-MaxI*)
fix d
assume $d: d \in \text{insert } 0 \text{ } (\text{max-depth-sort } ' (\text{set } \sigma s \cup \text{snd } ' (tv - \{x\})))$
show $\exists d' \in \text{max-depth-sort } ' \text{snd } ' \text{insert } x \text{ tv. } d \leq d'$
proof (*cases d \in max-depth-sort ' set \sigma s*)
case *True*
then obtain σ **where** *mem: \sigma \in set \sigma s* **and** $d: d = \text{max-depth-sort } \sigma$ **by**
auto
from *max-depth-sort-Fun[OF f mem assms(2)]*
show *?thesis* **unfolding** d **by** *auto*
qed (*insert d, auto*)
qed (*insert fin, auto*)
finally show *?thesis* .
qed
qed

lemma *max-depth-sort-le*: **assumes** $\text{tvars-smp } (mset2 \text{ mp}) \subseteq \text{tvars-smp } (mset2 \text{ mp}')$
shows $\text{max-depth-sort-smp } mp \leq \text{max-depth-sort-smp } mp'$
unfolding *max-depth-sort-smp-def*
apply (*rule Max-le-MaxI*)
apply *force*
apply *force*
apply *force*
using *assms* **by** *auto*

lemma *mp-step-tvars*: $mp \rightarrow_{ss} mp' \implies \text{tvars-smp } (mset2 \text{ mp}') \subseteq \text{tvars-smp } (mset2 \text{ mp})$
proof (*induct rule: smp-step-ms.induct*)
case $*$: (*smp-decomp eqc f n eqcn mp*)
from $*(2)$ **show** *?case*
proof (*clarsimp, goal-cases*)
case ($1 \ x \ s \ i \ ti$)
from $*(1)[OF \ 1(4)] \ 1(2,3)$
show *?case* **by** (*intro bexI[OF - 1(4)], cases ti, auto*)
qed
qed *auto*

lemma *max-depth-sort-p-inst*: **assumes** $\text{mem: } \{\#\{\# \text{Var } x, t\#\}\# \} \in \# \ p$

```

and  $t$ : is-Fun  $t$ 
and  $\tau$ :  $\tau \in \tau s n x$ 
and  $fin$ : finite-constructor-form-pat (mset3  $p$ )
shows max-depth-sort-p  $p > \text{max-depth-sort-p}$  (image-mset (image-mset (image-mset
( $\lambda t. t \cdot \tau$ )))  $p$ )
proof –
  obtain  $mp$   $p'$  where  $p$ :  $p = \text{add-mset}$   $mp$   $p'$  and  $mp$ :  $mp = \{\#\{\# \text{Var } x, t\}\#\}$ 

  using mset-add[OF mem, of thesis] by simp
from  $fin$ [unfolded finite-constructor-form-pat-def  $p$ ]
have  $fin$ : finite-constructor-form-mp (mset2  $mp$ ) by auto
show ?thesis unfolding  $p$ 
proof (simp add: max-depth-sort-p-def image-mset.compositionality o-def)
  show max-depth-sort-smp (image-mset (image-mset ( $\lambda t. t \cdot \tau$ ))  $mp$ ) +
    ( $\sum_{x \in \#p'}. \text{max-depth-sort-smp}$  (image-mset (image-mset ( $\lambda t. t \cdot \tau$ ))  $x$ ))
     $< \text{max-depth-sort-smp}$   $mp + \sum \#$  (image-mset max-depth-sort-smp  $p'$ ) (is  $?a1$ 
+  $?b1 < ?a2 + ?b2$ )
  proof (rule add-less-le-mono)
    show  $?a1 < ?a2$ 
      by (rule mds-decrease-inst[OF t tau mp fin])
    show  $?b1 \leq ?b2$ 
      proof (rule sum-mset-mono, rule mds-weak-decrease-inst[OF tau])
        from  $fin$ [unfolded finite-constructor-form-mp-def  $mp$ ] obtain  $\iota$ 
          where  $fin$ : finite-sort  $C \iota$  and  $x$ : Var  $x : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid \iota SS)$ 
          by auto
        from  $x$   $fin$  show finite-sort  $C$  (snd  $x$ )
        by (auto simp: hastype-def restrict-map-def split: if-splits)
      qed
    qed
  qed

```

```

lemma depth-gterm-le-card: finite-sort  $C \sigma \implies t : \sigma$  in  $\mathcal{T}(D) \implies D \subseteq_m C \implies$ 
depth-gterm  $t \leq \text{card}$  (dom  $D$ )
proof (induct t arbitrary: \sigma D)
  case *: (Fun  $f$   $ts \sigma D$ )
    from  $*(3)$ [unfolded Fun-hastype] obtain  $\sigma s$  where  $f$ :  $f : \sigma s \rightarrow \sigma$  in  $D$  and  $ts$ :
 $ts :_l \sigma s$  in  $\mathcal{T}(D)$  by auto
    from  $*(4)$  have dom  $D \subseteq \text{dom } C$  by (rule map-le-implies-dom-le)
    with finite-C have  $fin$ : finite (dom  $D$ ) by (metis finite-subset)
    from  $f$  have  $mem$ :  $(f, \sigma s) \in \text{dom } D$  by auto
    define  $\text{dom} D'$  where  $\text{dom} D' = \text{dom } D - \{(f, \sigma s)\}$ 
    define  $D'$  where  $D' = D \mid \iota \text{dom} D'$ 
    have  $\text{dom}$ : dom  $D = \text{insert}$   $(f, \sigma s)$  (dom  $D'$ ) unfolding  $D'$ -def  $\text{dom} D'$ -def using
 $mem$  by auto
    from arg-cong[OF this, of card]
    have  $\text{card} D$ : card (dom  $D$ ) = Suc (card (dom  $D'$ ))
    unfolding  $D'$ -def  $\text{dom} D'$ -def using  $mem$   $fin$  by auto
    from  $f$   $*(4)$  have  $f : \sigma s \rightarrow \sigma$  in  $C$  by (metis subssigD)

```

hence $\text{sig}: \sigma \in S$ **using** $C\text{-sub-}S$ **by** blast
show $?case$
proof ($cases\ ts$)
 case Nil
 thus $?thesis$ **unfolding** $\text{card}D$ **by** simp
next
 case ($Cons\ t\ ts'$)
 hence $\text{map}\ \text{depth-gterm}\ ts \neq []$ **by** auto
 from $\text{max-list-mem}[OF\ \text{this}]$ **obtain** t **where** $t: t \in \text{set}\ ts$
 and $\text{max-list}\ (\text{map}\ \text{depth-gterm}\ ts) = \text{depth-gterm}\ t$ **by** auto
 hence $\text{depth}: \text{depth-gterm}\ (\text{Fun}\ f\ ts) = \text{Suc}\ (\text{depth-gterm}\ t)$ **by** simp
 note $IH = *(1)[OF\ t]$
 have $D' \subseteq_m C$ **using** $*(4)$ **unfolding** $D'\text{-def}$
 using map-le-trans **by** blast
 note $IH = IH[OF\ -\ -\ \text{this}]$
 from $t\ ts$ **obtain** τ **where** $\text{tau}: \tau \in \text{set}\ \sigma s$ **and** $\text{tt}: t : \tau$ **in** $\mathcal{T}(D)$
 by ($\text{force}\ \text{simp}: \text{list-all2-conv-all-nth}\ \text{set-conv-nth}$)
 have $f : \sigma s \rightarrow \sigma$ **in** C **by** fact
 from $\text{finite-arg-sort}[OF\ *(2)\ \text{this}\ \text{tau}]$ **have** $\text{finite-sort}\ C\ \tau$ **by** auto
 note $IH = IH[OF\ \text{this}]$
 have $t : \tau$ **in** $\mathcal{T}(D')$
 proof ($\text{rule}\ \text{ccontr}$)
 assume $\neg ?thesis$
 from $\text{tt}\ \text{this}$ **have** $\exists s. t \triangleright s \wedge s : \sigma$ **in** $\mathcal{T}(D)$
 proof induct
 case ($\text{Fun}\ g\ ss\ \tau s\ \tau$)
 show $?case$
 proof ($cases\ ss :_l\ \tau s$ **in** $\mathcal{T}(D')$)
 case $True$
 from $\text{this}\ \text{Fun}(4)$ **have** $\neg (g : \tau s \rightarrow \tau$ **in** $D')$
 by ($\text{metis}\ \text{Fun-hastypeI}$)
 with $\text{Fun}(1)$ **have** $(g, \tau s) = (f, \sigma s)$ **unfolding** $D'\text{-def}\ \text{dom}D'\text{-def}$
 by ($\text{auto}\ \text{simp}: \text{restrict-map-def}\ \text{fun-hastype-def}\ \text{hastype-def}\ \text{split}: \text{if-splits}$)
 with $\text{Fun}(1)\ f$ **have** $\tau = \sigma$ **by** ($\text{simp}\ \text{add}: \text{fun-has-same-type}$)
 from $\text{Fun}(1,2)[\text{unfolded}\ \text{this}]$ **have** $\text{Fun}\ g\ ss : \sigma$ **in** $\mathcal{T}(D)$ **by** ($\text{rule}\ \text{Fun-hastypeI}$)
 thus $?thesis$ **by** blast
 next
 case $False$
 with $\text{Fun}(2)$ **obtain** i **where** $i: i < \text{length}\ ss$
 and $\text{ssi}: \neg (ss ! i : \tau s ! i$ **in** $\mathcal{T}(D')$)
 by ($\text{force}\ \text{simp}: \text{list-all2-conv-all-nth}\ \text{set-conv-nth}$)
 from $\text{list-all2-nth}D[OF\ \text{Fun}(3)\ i, \text{rule-format}, OF\ \text{ssi}]$
 obtain s **where** $s \trianglelefteq ss ! i$ **and** $s: s : \sigma$ **in** $\mathcal{T}(D)$
 by auto
 show $?thesis$
 proof ($\text{intro}\ \text{exI}[of\ -\ s]\ \text{conjI}\ s$)
 have $s \trianglelefteq ss ! i$ **by** fact
 also **have** $ss ! i \trianglelefteq \text{Fun}\ g\ ss$ **using** i **by** simp

```

      finally show  $s \leq \text{Fun } g \text{ ss}$  by auto
    qed
  qed
  qed auto
  then obtain  $s$  where  $s \leq t$  and  $s : s : \sigma$  in  $\mathcal{T}(D)$  by auto
  with  $t$  have  $\text{sub} : \text{Fun } f \text{ ts} \triangleright s$  by auto
  from  $s$  *(4) have  $s : s : \sigma$  in  $\mathcal{T}(C)$  by (metis hastype-in-Term-mono-left)
  from *(3,4) have  $t : \text{Fun } f \text{ ts} : \sigma$  in  $\mathcal{T}(C)$  by (metis hastype-in-Term-mono-left)
  from  $\text{sub}$  obtain  $c$  where  $c \neq \square$  and  $\text{Fun } f \text{ ts} = c \langle s \rangle$  by blast
  from  $\text{apply-ctxt-hastype-imp-hastype-context}[OF \ t[\text{unfolded this}]] \ s$ 
  have  $c : c : \sigma \rightarrow \sigma$  in  $\mathcal{C}(C, \lambda x. \text{None})$  by auto
  from  $\text{max-depth-sort-wit}[OF \ *(2) \ \text{sig}]$ 
  obtain  $t$  where  $t : t : \sigma$  in  $\mathcal{T}(C)$  and  $\text{large} : \bigwedge t'. t' : \sigma \text{ in } \mathcal{T}(C) \implies$ 
 $\text{depth-gterm } t' \leq \text{depth-gterm } t$ 
  by auto
  from  $t \ c$  have  $c \langle t \rangle : \sigma$  in  $\mathcal{T}(C)$  by (metis  $\text{apply-ctxt-hastype}$ )
  from  $\text{large}[OF \ \text{this}]$  have  $\text{contra} : \text{depth-gterm } (c \langle t \rangle) \leq \text{depth-gterm } t$  .
  from  $\langle c \neq \square \rangle$  obtain  $f \ \text{bef } d \ \text{aft}$  where  $c : c = \text{More } f \ \text{bef } d \ \text{aft}$  by (cases  $c$ ,
  auto)
  have  $\text{depth-gterm } t \leq \text{depth-gterm } (d \langle t \rangle)$ 
  proof (induct  $d$ )
    case (More  $f \ \text{bef } d \ \text{aft}$ )
      thus ?case using  $\text{max-list}[of \ \text{depth-gterm } (d \langle t \rangle) \ \text{map } \text{depth-gterm } \ \text{bef} \ @$ 
 $\text{depth-gterm } \ d \langle t \rangle \ \# \ \text{map } \text{depth-gterm } \ \text{aft}]$ 
      by auto
    qed auto
    also have  $\dots < \text{depth-gterm } (c \langle t \rangle)$  unfolding  $c$ 
      using  $\text{max-list}[of \ \text{depth-gterm } (d \langle t \rangle) \ \text{map } \text{depth-gterm } \ \text{bef} \ @ \ \text{depth-gterm}$ 
 $d \langle t \rangle \ \# \ \text{map } \text{depth-gterm } \ \text{aft}]$ 
      by auto
    also have  $\dots \leq \text{depth-gterm } t$  by (rule  $\text{contra}$ )
    finally show  $\text{False}$  by simp
  qed
  from  $\text{IH}[OF \ \text{this}]$  show ?thesis unfolding  $\text{depth cardD}$  by auto
  qed
  qed auto

```

```

lemma  $\text{max-depth-sort-smp-le-card}$ : assumes  $\text{finite-constructor-form-mp}$  ( $\text{mset2}$ 
 $\text{mp}$ )
  shows  $\text{max-depth-sort-smp } \text{mp} \leq \text{card } (\text{dom } C)$ 
  unfolding  $\text{max-depth-sort-smp-def}$ 
  proof (subst  $\text{Max-le-iff}$ , force intro!:  $\text{finite-imageI}$ , force, intro ballI)
    fix  $d$ 
    assume  $d : d \in \text{insert } 0 \ (\text{max-depth-sort } \ ' \ \text{snd } \ ' \ \bigcup \ (\bigcup \ ((\ ' \ \text{vars } \ ' \ \text{mset2 } \ \text{mp})))$ )
    show  $d \leq \text{card } (\text{dom } C)$ 
    proof (cases  $d = 0$ )
      case  $\text{False}$ 
        with  $d$  obtain  $\text{eqc } t \ x$  where  $\text{eqc} \in \text{mset2 } \ \text{mp} \ t \in \text{eqc}$  and

```

```

    x: x ∈ vars t and d: d = max-depth-sort (snd x)
  by blast
  with assms[unfolded finite-constructor-form-mp-def] obtain ι where fin: fi-
nite-sort C ι
  and t: t : ι in T(C, V |' SS) by force
  define σ where σ = snd x
  from t x have inS: σ ∈ S
  by induct (auto simp: restrict-map-def hastype-def list-all2-conv-all-nth set-conv-nth

    σ-def split: if-splits)
  from t fin x have fin: finite-sort C σ
  proof (induct)
    case (Var v σ)
    then show ?case by (auto simp: restrict-map-def hastype-def σ-def split:
if-splits)
  next
    case *: (Fun f ss σ s τ)
    from finite-arg-sort[OF *(4,1)] *(2,3,5)
    show ?case by (auto simp: list-all2-conv-all-nth set-conv-nth)
  qed
  from max-depth-sort-wit[OF this inS, folded d[folded σ-def]]
  obtain t where tσ: t : σ in T(C) depth-gterm t = d
  ∧ t'. t' : σ in T(C) ⇒ depth-gterm t' ≤ d by auto
  from depth-gterm-le-card[OF fin tσ(1), unfolded tσ] show ?thesis by simp
  qed auto
  qed

```

```

lemma max-depth-sort-p-le-card-size: assumes finite-constructor-form-pat (mset3
p)
shows max-depth-sort-p p ≤ card (dom C) * size p
proof -
  have max-depth-sort-p p = sum-mset (image-mset max-depth-sort-smp p) un-
folding max-depth-sort-p-def ..
  also have ... ≤ sum-mset (image-mset (λ mp. card (dom C)) p)
  by (rule sum-mset-mono, rule max-depth-sort-smp-le-card,
insert assms, auto simp: finite-constructor-form-pat-def)
  finally show ?thesis by (simp add: ac-simps)
  qed

```

```

definition num-syms-smp :: ('f, 's)simple-match-problem-ms ⇒ nat where
  num-syms-smp mp = sum-mset (image-mset num-syms (sum-mset mp))

```

```

lemma num-syms-subset: assumes sum-mset mp ⊂# sum-mset mp'
shows num-syms-smp mp < num-syms-smp mp'
proof -
  from assms obtain ts where sum-mset mp' = ts + sum-mset mp and ts ≠ {#}

  by (metis add.commute subset-mset.lessE)

```

thus *?thesis unfolding num-syms-smp-def*
by (*cases ts, auto*)
qed

definition *max-dupl-smp* **where**
 $max-dupl-smp\ mp = Max\ (insert\ 0\ ((\lambda\ x.\ (\sum\ t \in \#\ sum-mset\ mp.\ count\ (syms-term\ t)\ (Inl\ x))))\ 'tvars-smp\ (mset2\ mp))$

definition *max-dupl-p* $::\ ('f, 's)simple-pat-problem-ms \Rightarrow nat$ **where**
 $max-dupl-p\ p = sum-mset\ (image-mset\ max-dupl-smp\ p)$

lemma *max-dupl-smp*: $(\sum\ t \in \#\ sum-mset\ mp.\ count\ (syms-term\ t)\ (Inl\ x)) \leq max-dupl-smp\ mp$

proof (*cases x \in tvars-smp (mset2 mp)*)
case *True*
thus *?thesis unfolding max-dupl-smp-def*
by (*intro Max-ge[OF finite.insertI[OF finite-imageI]], auto*)

next

case *False*
have $(\sum\ t \in \#\sum\ \#\ mp.\ count\ (syms-term\ t)\ (Inl\ x)) = 0$
proof (*clarsimp*)
fix *eq t*
assume $eq \in \#\ mp\ t \in \#\ eq$
with *False* **have** $x \notin vars\ t$ **by** *auto*
thus $count\ (syms-term\ t)\ (Inl\ x) = 0$ **unfolding** *vars-term-syms-term*
by (*simp add: count-eq-zero-iff*)

qed

thus *?thesis by linarith*

qed

lemma *max-dupl-smp-le-num-syms*: $max-dupl-smp\ mp \leq num-syms-smp\ mp$
unfolding *max-dupl-smp-def*

proof (*subst Max-le-iff, force intro!: finite-imageI simp: tvars-match-def, force, intro ballI*)

fix *n*

assume $n: n \in insert\ 0\ \{\sum\ t \in \#\sum\ \#\ mp.\ count\ (syms-term\ t)\ (Inl\ x)\ |\ x \in tvars-smp\ (mset2\ mp)\}$

show $n \leq num-syms-smp\ mp$

proof (*cases n = 0*)

case *False*

with *n* **obtain** *x* **where** $x: x \in tvars-smp\ (mset2\ mp)$

and $n: n = (\sum\ t \in \#\sum\ \#\ mp.\ count\ (syms-term\ t)\ (Inl\ x))$ **by** *auto*

have $nt: num-syms-smp\ mp = \sum\ \#\ (image-mset\ num-syms\ (\sum\ \#\ mp))$

unfolding *num-syms-smp-def ..*

show *?thesis unfolding n nt*

proof (*rule sum-mset-mono, goal-cases*)

case (*1 t*)

show $count\ (syms-term\ t)\ (Inl\ x) \leq num-syms\ t$

unfolding *num-syms-def* **by** (*rule count-le-size*)
qed
qed *auto*
qed

lemma *num-syms-smp- τ s*: **assumes** $\tau: \tau \in \tau s n x$
shows $\text{num-syms-smp } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau)) mp) \leq \text{num-syms-smp } mp + \text{max-dupl-smp } mp * m$
proof –
have $\text{num-syms-smp } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau)) mp) =$
 $(\sum eq \in \#mp. (\sum t \in \#eq. \text{num-syms } (t \cdot \tau)))$
unfolding *num-syms-smp-def image-mset.compositionality o-def*
by (*induct mp, auto simp: image-mset.compositionality o-def*)
also have $\dots \leq (\sum eq \in \#mp. (\sum t \in \#eq. \text{num-syms } t + \text{count } (\text{syms-term } t)$
 $(\text{Inl } x) * m))$
using *num-syms- τ s[OF τ]*
by (*intro sum-mset-mono, auto*)
also have $\dots = (\sum eq \in \#mp. (\sum t \in \#eq. \text{num-syms } t)) + (\sum eq \in \#mp. (\sum$
 $t \in \#eq. \text{count } (\text{syms-term } t) (\text{Inl } x) * m))$
unfolding *sum-mset.distrib* **by** *simp*
also have $(\sum eq \in \#mp. (\sum t \in \#eq. \text{num-syms } t)) = \text{num-syms-smp } mp$
unfolding *num-syms-smp-def* **by** (*induct mp, auto*)
also have $(\sum eq \in \#mp. (\sum t \in \#eq. \text{count } (\text{syms-term } t) (\text{Inl } x) * m))$
 $= (\sum eq \in \#mp. (\sum t \in \#eq. \text{count } (\text{syms-term } t) (\text{Inl } x))) * m$
unfolding *sum-mset-distrib-right* **by** *simp*
also have $(\sum eq \in \#mp. (\sum t \in \#eq. \text{count } (\text{syms-term } t) (\text{Inl } x)))$
 $= (\sum t \in \# \text{sum-mset } mp. \text{count } (\text{syms-term } t) (\text{Inl } x))$
by (*induct mp, auto*)
also have $\dots * m \leq \text{max-dupl-smp } mp * m$
using *max-dupl-smp[of $x mp$]* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *max-dupl-smp- τ s*: **assumes** $\tau: \tau \in \tau s n x$
and *disj: fst ' tvars-smp (mset2 mp) $\cap \{n..<n + m\} = \{\}$*
shows $\text{max-dupl-smp } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau)) mp) \leq \text{max-dupl-smp } mp$
proof –
from τ [*unfolded τ s-def*] **obtain** *f ss*
where *f: f : ss \rightarrow snd x in C and $\tau: \tau = \tau c n x (f, ss)$ by *auto*
define *vs* **where** *vs = indexed-from n ss*
define *s* **where** *s = (Fun f (map Var vs))*
have *tau: $\tau = \text{subst } x s$* **unfolding** τ *τc -def s -def vs -def* **by** *auto*
show *?thesis*
proof (*cases $x \in \text{tvars-smp } (\text{mset2 } mp)$*)
case *False*
have $\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau)) mp = \text{image-mset } (\text{image-mset } (\lambda t.$
 $t \cdot \text{Var})) mp$
proof (*intro image-mset-cong term-subst-eq, goal-cases*)*

```

    case (1 eq t y)
    with False show ?case by (auto simp: tau subst-def)
  qed
  thus ?thesis by simp
next
case x: True
show ?thesis unfolding max-dupl-smp-def
proof ((intro Max-le-MaxI; (intro finite.insertI, force)?), force, goal-cases)
  case (1 d)
  show ?case
  proof (cases d = 0)
    case False
    with 1 obtain y where
      d: d = ( $\sum t \in \# \sum \#$  (image-mset (image-mset ( $\lambda t. t \cdot \tau$ )) mp). count
      (syms-term t) (Inl y))
      and y: y  $\in$  tvars-smp (mset2 (image-mset (image-mset ( $\lambda t. t \cdot \tau$ )) mp))
  by auto
  have syms-s: syms-term s = add-mset (Inr f) (mset (map Inl vs)) unfolding
  s-def
  by (simp, induct vs, auto)
  define repl :: ('f, nat  $\times$  's) Term.term  $\Rightarrow$  (nat  $\times$  's + 'f) multiset
  where repl t = replicate-mset (count (syms-term t) (Inl x)) (Inl x) for t
  let ?add = ( $\sum t \in \# \sum \#$  mp. count (repl t) (Inl y))
  let ?symsy = ( $\sum t \in \# \sum \#$  mp. count (syms-term t) (Inl y))
  let ?symsxy = ( $\sum t \in \# \sum \#$  mp. count (syms-term t) (Inl x) * count
  (syms-term s) (Inl y))
  let ?symsx = ( $\sum t \in \# \sum \#$  mp. count (syms-term t) (Inl x))
  have d = ( $\sum t \in \# \sum \#$  mp. count (syms-term (t  $\cdot$   $\tau$ )) (Inl y))
  unfolding d
  proof (induct mp, auto, goal-cases)
    case (1 eq mp)
    show ?case by (induct eq, auto)
  qed
  also have ...  $\leq$  ... + ?add by simp
  also have ... =
    ( $\sum t \in \# \sum \#$  mp. count (syms-term (t  $\cdot$   $\tau$ ) + repl t) (Inl y))
  unfolding sum-mset.distrib[symmetric]
  by (rule arg-cong[of - - sum-mset], rule image-mset-cong, simp)
  also have ... = ?symsy + ?symsxy
  unfolding repl-def tau syms-term-subst sum-mset.distrib[symmetric] by
simp
  finally have d: d  $\leq$  ?symsy + ?symsxy .
  show ?thesis
  proof (cases y  $\in$  set vs)
    case False
    hence ?symsxy = 0 unfolding syms-s by auto
    with d have d: d  $\leq$  ?symsy by auto
    from y[simplified] obtain eq t
    where eq: eq  $\in$  # mp and t: t  $\in$  # eq and y: y  $\in$  vars (t  $\cdot$   $\tau$ )

```

```

      by auto
      from False this[unfolded vars-term-subst tau subst-def s-def]
      have  $y \in \text{vars } t$  by (auto split: if-splits)
      hence  $y \in \text{tvars-smp } (mset2 \ mp)$  using eq t by auto
      with d show ?thesis by auto
    next
      case True
      have dist: distinct vs
        by (simp add: vs-def)
      from split-list[OF True] obtain vs1 vs2 where vs: vs = vs1 @ y # vs2
    by auto
      with dist have nmem:  $y \notin \text{set } vs1 \ y \notin \text{set } vs2$  by auto
      have count (syms-term s) (Inl y) = 1 unfolding syms-s vs using nmem
        by (auto simp: count-eq-zero-iff)
      with d have d:  $d \leq ?symsy + ?symsx$  by auto
      from True have fst  $y \in \{n..<n + m\}$  using m[OF f]
        by (simp add: in-set-indexed-from-eq vs-def)
      with disj have  $y \notin \text{tvars-smp } (mset2 \ mp)$  by blast
      hence ?symsy = 0
        by (auto simp: count-eq-zero-iff vars-term-syms-term)
      with d have  $d \leq ?symsx$  by auto
      with x show ?thesis by auto
    qed
  qed auto
  qed
  qed
  qed

```

definition *num-syms-p* :: $(f, s)\text{simple-pat-problem-ms} \Rightarrow \text{nat}$ **where**
num-syms-p p = *sum-mset (image-mset num-syms-smp p)*

lemma *num-syms-p-add[simp]*: *num-syms-p (add-mset mp p) = num-syms-smp mp + num-syms-p p*
unfolding *num-syms-p-def* **by** *simp*

lemma *max-dupl-p-le-num-syms*: *max-dupl-p p ≤ num-syms-p p*
unfolding *max-dupl-p-def num-syms-p-def*
by (rule *sum-mset-mono*[OF *max-dupl-smp-le-num-syms*])

lemma *max-dupl-p-add[simp]*: *max-dupl-p (add-mset mp p) = max-dupl-smp mp + max-dupl-p p*
unfolding *max-dupl-p-def* **by** *simp*

lemma *max-dupl-mono-main*: **assumes** $\bigwedge x. (\sum t \in \# \sum \# mp. \text{count } (\text{syms-term } t) (\text{Inl } x)) \leq (\sum t \in \# \sum \# mp'. \text{count } (\text{syms-term } t) (\text{Inl } x))$
and *tvars-smp (mset2 mp) ⊆ tvar-smp (mset2 mp')*
shows *max-dupl-smp mp ≤ max-dupl-smp mp'*
unfolding *max-dupl-smp-def*

```

proof ((intro Max-le-MaxI; (intro finite.insertI, force)?), force, goal-cases)
  case (1 d)
  show ?case
  proof (cases d = 0)
    case False
    with 1 obtain x where d: d = ( $\sum t \in \# \sum \# mp. \text{count} (\text{syms-term } t) (\text{Inl } x)$ )

    and x: x  $\in$  tvars-smp (mset2 mp) by auto
    with x assms(2) have x: x  $\in$  tvars-smp (mset2 mp') by blast
    define d' where d' = ( $\sum t \in \# \sum \# mp'. \text{count} (\text{syms-term } t) (\text{Inl } x)$ )
    have d  $\leq$  d' unfolding d d'-def by fact
    with x show ?thesis unfolding d'-def by blast
  qed auto
qed

```

```

lemma max-dupl-mono: assumes sum-mset mp  $\subseteq \#$  sum-mset mp'
  shows max-dupl-smp mp  $\leq$  max-dupl-smp mp'
proof (rule max-dupl-mono-main)
  from set-mp[OF set-mset-mono[OF assms]]
  show tvars-smp (mset2 mp)  $\subseteq$  tvars-smp (mset2 mp') by force
  from assms obtain mp2 where eq: sum-mset mp' = sum-mset mp + mp2 by
(rule subset-mset.less-eqE)
  fix x
  show ( $\sum t \in \# \sum \# mp. \text{count} (\text{syms-term } t) (\text{Inl } x)$ )  $\leq$  ( $\sum t \in \# \sum \# mp'. \text{count} (\text{syms-term } t) (\text{Inl } x)$ )
    unfolding eq image-mset-union by auto
qed

```

```

definition syms-smp :: ('f, 's)simple-match-problem-ms  $\Rightarrow$  (nat  $\times$  's + 'f) multiset
where
  syms-smp mp = sum-mset (image-mset syms-term (sum-mset mp))

```

```

definition syms-ecq :: ('f, nat  $\times$  's)term multiset  $\Rightarrow$  (nat  $\times$  's + 'f) multiset where
  syms-ecq eqc = sum-mset (image-mset syms-term eqc)

```

```

lemma syms-smp-to-ecq: syms-smp mp = sum-mset (image-mset syms-ecq mp)
  unfolding syms-smp-def syms-ecq-def
  by (induct mp, auto)

```

```

lemma nums-eq-size-syms-smp: num-syms-smp mp = size (syms-smp mp)
  unfolding num-syms-smp-def syms-smp-def num-syms-def
proof (induct mp)
  case (add eq mp)
  show ?case
    apply (simp add: add)
    apply (induct eq, auto)
  done

```

qed *auto*

lemma *num-syms-sym-subset*: **assumes** *syms-smp mp* $\subset\#$ *syms-smp mp'*
shows *num-syms-smp mp* < *num-syms-smp mp'*

proof –

from *assms* **obtain** *d* **where** *mp'*: *syms-smp mp' = d + syms-smp mp* **and** *d*:
d \neq $\{\#\}$

by (*metis add commute subset-mset.lessE*)

thus *?thesis* **unfolding** *nums-eq-size-syms-smp* **by** (*cases d, auto*)

qed

lemma *num-syms-smp-step*: *mp* \rightarrow_{ss} *mp'* \implies *finite-constructor-form-mp* (*mset2 mp*)

\implies *num-syms-smp mp'* < *num-syms-smp mp*

proof (*induct rule: smp-step-ms.induct*)

case (*smp-dup t eqc mp*)

show *?case* **by** (*rule num-syms-subset, auto*)

next

case (*smp-singleton t mp*)

show *?case* **by** (*rule num-syms-subset, auto*)

next

case (*smp-triv-sort t ι eq mp*)

show *?case* **by** (*rule num-syms-subset, auto*)

next

case *: (*smp-decomp eqc f n eqcn mp*)

define *nums* **where** *nums = mset-set* $\{0..<n\}$

from *(4) **have** *eqc*: *eqc* \neq $\{\#\}$ **unfolding** *finite-constructor-form-mp-def* **by**
auto

define *arg* **where** *arg t =* $(\sum i \in \#nums. \text{syms-term } (\text{args } t ! i))$ **for** *t* :: (*f, nat*
 \times *'s*) *term*

{

fix *t*

assume *t* $\in\#$ *eqc*

from *(1)[*OF this*] **obtain** *ts* **where** *t*: *t = Fun f ts* **and** *len*: *n = length ts*

by (*cases t, auto*)

have *id*: *mset ts = image-mset* ($(!) ts$) (*mset* $[0..<\text{length } ts]$)

by (*metis map-nth' mset-map*)

have *syms-term t = add-mset* (*Inr f*) (*arg t*) **unfolding** *t arg-def nums-def len*

by (*auto simp: id image-mset.compositionality o-def*)

} **note** *step = this*

have *?case* \iff *num-syms-smp eqcn* < *num-syms-smp* $\{\#eqc\#$ **using** * **un-**
folding *num-syms-smp-def* **by** *auto*

also have ...

proof (*rule num-syms-sym-subset*)

from *eqc* **obtain** *t eqc'* **where** *eqc*: *eqc = add-mset t eqc'* **by** (*cases eqc, auto*)

have *syms-smp eqcn =* $(\sum i \in \#nums. \text{syms-eq } \{\#\text{args } t ! i. t \in\# eqc\#$)

unfolding *image-mset.compositionality o-def nums-def syms-smp-to-eqc* *(2)

by *simp*

also have ... = $(\sum t \in\#eqc. \text{arg } t)$ **unfolding** *arg-def*

unfolding *syms-eqc-def image-mset.compositionality o-def*
by (*rule sum-mset.swap*)
also have $\dots = \arg t + (\sum t \in \#eqc'. \arg t)$ **unfolding** *eqc* **by** *auto*
also have $\dots \subset \# \text{syms-term } t + (\sum t \in \#eqc'. \text{syms-term } t)$
proof (*rule subset-mset.add-less-le-mono*)
from *step[of t] eqc*
show $\arg t \subset \# \text{syms-term } t$ **by** *auto*
from *step* **have** $t \in \# eqc' \implies \text{syms-term } t = \text{add-mset } (\text{Inr } f) (\arg t)$
for *t* **unfolding** *eqc* **by** *auto*
have $\sum \# (\text{image-mset } \text{syms-term } eqc') = \sum \# (\text{image-mset } (\lambda t. \text{add-mset } (\text{Inr } f) (\arg t)) eqc')$
by (*subst image-mset-cong[OF step], auto*)
also have $\dots = \sum \# (\text{image-mset } (\lambda t. \arg t) eqc') + \text{image-mset } (\lambda t. \text{Inr } f) eqc'$
by (*induct eqc', auto*)
finally
show $\sum \# (\text{image-mset } \arg eqc') \subseteq \# \sum \# (\text{image-mset } \text{syms-term } eqc')$ **by**
simp
qed
also have $\text{syms-term } t + (\sum t \in \#eqc'. \text{syms-term } t) = (\sum t \in \#eqc. \text{syms-term } t)$
unfolding *eqc* **by** *simp*
also have $\dots = \text{syms-smp } \{\#eqc\#}$ **unfolding** *syms-smp-to-eqc syms-eqc-def*
by *simp*
finally show $\text{syms-smp } eqcn \subset \# \text{syms-smp } \{\#eqc\#}$.
qed
finally show *?case* .
qed

lemma *num-syms-smp-pos[simp]*: **assumes** *finite-constructor-form-mp (mset2 mp)*

and $mp \neq \{\#\}$
shows *num-syms-smp mp > 0*
proof –
from *assms* **obtain** *eqc mp'* **where** $mp: mp = \text{add-mset } eqc mp'$ **by** (*cases mp, auto*)
with *assms* **have** $eqc \neq \{\#\}$ **unfolding** *finite-constructor-form-mp-def* **by** *auto*
thus *?thesis* **unfolding** *mp num-syms-smp-def* **by** (*cases eqc, auto*)
qed

lemma *count-syms-smp*: $(\sum t \in \# \sum \# mp. \text{count } (\text{syms-term } t) (\text{Inl } x)) = (\text{count } (\text{syms-smp } mp) (\text{Inl } x))$

unfolding *syms-smp-def*
proof (*induct mp, auto simp: image-mset.compositionality o-def, goal-cases*)
case (*1 eq mp*)
show *?case* **by** (*induct eq, auto*)
qed

lemma *max-dupl-smp-step*: $mp \rightarrow_{ss} mp' \implies \text{max-dupl-smp } mp' \leq \text{max-dupl-smp } mp$

```

mp
proof (induct rule: smp-step-ms.induct)
  case (smp-dup t eqc mp)
  show ?case by (rule max-dupl-mono, auto)
next
  case (smp-singleton t mp)
  show ?case by (rule max-dupl-mono, auto)
next
  case (smp-triv-sort t ι eq mp)
  show ?case by (rule max-dupl-mono, auto)
next
  case *: (smp-decomp eqc f n eqcn mp)
  show ?case
proof (rule max-dupl-mono-main, unfold count-syms-smp, rule mset-subset-eq-count)
  have syms-smp eqcn ⊆# syms-smp {#eqc#}
  proof -
    define nums where nums = mset-set {0..<n}
    define arg where arg t = (∑ i∈#nums. syms-term (args t ! i)) for t :: ('f,
nat × 's) term
    have syms-smp eqcn = (∑ i∈#nums. syms-eqc {#args t ! i. t ∈# eqc#})
      unfolding image-mset.compositionality o-def nums-def syms-smp-to-eqc
    *(2)
      by simp
    also have ... = (∑ t∈#eqc. arg t) unfolding arg-def
      unfolding syms-eqc-def image-mset.compositionality o-def
      by (rule sum-mset.swap)
    finally have transform: syms-smp eqcn = ∑ # (image-mset arg eqc) .

  have ?thesis ↔ ∑ # (image-mset arg eqc) ⊆# ∑ # (image-mset syms-term
eqc)
    unfolding transform unfolding syms-smp-def by simp
  also have ...
  proof -
    {
      fix t
      assume t ∈# eqc
      from *(1)[OF this] obtain ts where t: t = Fun f ts and len: length ts =
n by (cases t, auto)
      have id: mset ts = mset (map (λ i. ts ! i) [0..<length ts])
        by (intro arg-cong[of - - mset], rule nth-equalityI, auto)
      have arg t = (∑ i∈#mset-set {0..<length ts}. syms-term (ts ! i))
        unfolding arg-def t nums-def len by simp
      also have ... = (∑ # (image-mset syms-term (mset ts)))
        unfolding id
        by (auto simp: image-mset.compositionality o-def)
      finally have arg t ⊆# syms-term t unfolding t by simp
    }
  thus ?thesis
  by (induct eqc, auto intro: subset-mset.add-mono)

```

```

qed
finally show ?thesis .
qed
thus syms-smp (eqcn + mp)  $\subseteq$ # syms-smp (add-mset eqc mp)
  unfolding syms-smp-def by auto

  have  $\bigcup (\bigcup_{x \in \text{set-mset } eqcn. \text{vars } ' \text{set-mset } x} \subseteq \bigcup (\text{vars } ' \text{set-mset } eqc)$ 
unfolding *(2)
proof (clarsimp, goal-cases)
  case (1 x s i t)
  from *(1) 1 have root t = Some (f,n) by auto
  with 1 have (x,s)  $\in$  vars t by (cases t, auto)
  with 1 show ?case by auto
qed
thus tvars-smp (mset2 (eqcn + mp))  $\subseteq$  tvars-smp (mset2 (add-mset eqc mp))
  by force
qed
qed

definition measure-p :: ('f,'s) simple-pat-problem-ms  $\Rightarrow$  nat where
  measure-p p = max-depth-sort-p p * (max-dupl-p p * m + 1) + num-syms-p p

lemma measure-p-bound: assumes finite-constructor-form-pat (mset3 p)
shows measure-p p  $\leq$  card (dom C) * size p * (num-syms-p p + 1) * (m + 1)
proof –
  {
    assume num-syms-p p  $\neq$  0
    from this[unfolded num-syms-p-def] obtain mp where mp: mp  $\in$ # p and
num-syms-smp mp  $\neq$  0 by auto
    from this[unfolded num-syms-smp-def] obtain eqc t where eqc: eqc  $\in$ # mp
and t: t  $\in$ # eqc and num-syms t  $\neq$  0
    by auto
    from assms mp have finite-constructor-form-mp (mset2 mp)
    by (auto simp: finite-constructor-form-pat-def)
    from this[unfolded finite-constructor-form-mp-def] eqc t obtain  $\iota$  where t :  $\iota$ 
in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by auto
    hence t  $\cdot$   $\sigma g$  :  $\iota$  in  $\mathcal{T}(C)$ 
    by (metis  $\sigma g' \sigma g'$ -def  $\sigma g$ -def sorted-map-cong subst-has-same-type)
    then obtain t where t :  $\iota$  in  $\mathcal{T}(C)$  by auto
    then obtain f where f  $\in$  dom C by (induct, auto simp: fun-hastype-def)
    hence card (dom C)  $\neq$  0 size p  $\neq$  0 using finite-C mp by auto
  } note non-triv-p = this
have measure-p p  $\leq$  (card (dom C) * size p) * (num-syms-p p * m + 1) +
num-syms-p p
unfolding measure-p-def
by (intro add-right-mono mult-mono max-dupl-p-le-num-syms max-depth-sort-p-le-card-size
assms, auto)
also have ...  $\leq$  (card (dom C) * size p) * (num-syms-p p * m + 1) + (card
(dom C) * size p) * num-syms-p p

```

```

proof (cases num-syms-p p = 0)
  case False
    from non-triv-p[OF this] have card (dom C) * size p ≠ 0 by auto
    thus ?thesis by (cases card (dom C) * size p, auto)
  qed auto
also have ... = (card (dom C) * size p) * (num-syms-p p * m + 1 + num-syms-p
p)
  by (simp add: algebra-simps)
also have ... ≤ (card (dom C) * size p) * ((num-syms-p p + 1) * (m + 1))
  by (rule mult-left-mono, auto)
finally show measure-p p ≤ card (dom C) * size p * (num-syms-p p + 1) * (m
+ 1)
  by (simp add: algebra-simps)
qed

```

```

lemma measure-p-num-syms: assumes max-depth-sort-p p ≤ max-depth-sort-p p'
  and max-dupl-p p ≤ max-dupl-p p'
  and num-syms-p p < num-syms-p p'
shows measure-p p < measure-p p'
  unfolding measure-p-def
  by (intro add-le-less-mono mult-mono, insert assms, auto)

```

```

lemma spp-step-complexity-and-termination:
  assumes p ⇒ss Pn
  and finite-constructor-form-pat (mset3 p)
  and pn ∈# Pn
shows measure-p pn < measure-p p
  using assms
proof (induct)
  case *: (spp-simp mp mp' p)
  hence pn: pn = add-mset mp' p by simp
  let ?p = add-mset mp p
  from * have fin: finite-constructor-form-mp (mset2 mp)
  unfolding finite-constructor-form-pat-def by auto
  from num-syms-smp-step[OF *(1) fin]
  have n: num-syms-p pn < num-syms-p ?p unfolding pn by simp
  have d: max-depth-sort-p pn ≤ max-depth-sort-p ?p
  unfolding pn
  apply simp
  apply (rule max-depth-sort-le)
  apply (rule mp-step-tvars)
  by fact
  from max-dupl-smp-step[OF *(1)] *
  have m: max-dupl-p pn ≤ max-dupl-p ?p by simp
  from d n m show ?case by (intro measure-p-num-syms, auto)
next

```

```

case *: (spp-delete mp p)
hence pn: pn = p
  and fin: finite-constructor-form-mp (mset2 mp)
  unfolding finite-constructor-form-pat-def by auto
let ?p = add-mset mp p
have d: max-depth-sort-p pn ≤ max-depth-sort-p ?p unfolding pn max-depth-sort-p-def
by auto
from *(1) have mp ≠ {#} by (cases, auto)
from num-syms-smp-pos[OF fin this]
have n: num-syms-p pn < num-syms-p ?p unfolding pn by simp
have m: max-dupl-p pn ≤ max-dupl-p ?p unfolding pn by simp
from d n m show ?case by (intro measure-p-num-syms, auto)
next
case *: (spp-delete-large-sort n x ι eq mp t p)
define pd where pd = mset (map mp [0..<Suc n])
from *(5) have pn: pn = p by auto
let ?p = p + pd
have m: max-dupl-p p ≤ max-dupl-p ?p unfolding max-dupl-p-def by auto
have d: max-depth-sort-p p ≤ max-depth-sort-p ?p unfolding max-depth-sort-p-def
by auto
have num-syms-p p < num-syms-p p + 1 by auto
also have ... ≤ num-syms-p p + num-syms-p pd
proof (rule add-left-mono)
  have mem: mp n ∈# pd unfolding pd-def by auto
  from *(4) have finite-constructor-form-mp (mset2 (mp n))
    by (auto simp: finite-constructor-form-pat-def)
  from *(1)[OF le-refl] num-syms-smp-pos[OF this]
  have 1 ≤ num-syms-smp (mp n) by (cases mp n, auto)
  thus 1 ≤ num-syms-p pd using mem unfolding num-syms-p-def
  by (metis One-nat-def add-eq-0-iff-both-eq-0 less-Suc0 linorder-not-less multi-member-split
sum-mset.insert)
qed
finally have n: num-syms-p p < num-syms-p ?p unfolding num-syms-p-def by
auto
from n m d have measure-p p < measure-p (p + pd) by (metis measure-p-num-syms)
thus ?case unfolding pn pd-def by auto
next
case *: (spp-inst x t p n)
then obtain τ where pn: pn = image-mset (image-mset (image-mset (λt. t ·
τ))) p
  and τ: τ ∈ τs n x using τs-list by auto
from max-depth-sort-p-inst[OF *(1-2) τ *(4)] *(5)
have d: max-depth-sort-p pn < max-depth-sort-p p unfolding pn by auto
have num-syms-p pn = (∑ mp∈#p. num-syms-smp (image-mset (image-mset
(λt. t · τ)) mp))
  unfolding num-syms-p-def pn image-mset.compositionality o-def by simp
also have ... ≤ (∑ mp∈#p. num-syms-smp mp + max-dupl-smp mp * m)
  by (rule sum-mset-mono, rule num-syms-smp-τs[OF τ])
also have ... = num-syms-p p + max-dupl-p p * m

```

```

  unfolding sum-mset.distrib num-syms-p-def max-dupl-p-def sum-mset-distrib-right
by auto
  finally have n: num-syms-p pn ≤ num-syms-p p + max-dupl-p p * m .
  have m: max-dupl-p pn ≤ max-dupl-p p
    unfolding max-dupl-p-def pn image-mset.compositionality o-def
  proof (rule sum-mset-mono, rule max-dupl-smp-τs[OF τ], goal-cases)
    case (1 mp)
    thus ?case using *(3) by fastforce
  qed
  show ?case unfolding measure-p-def
    by (intro measure-expr-decrease n m d)
next
  case *: (spp-split mp x t eqc mp' p)
  let ?p = add-mset mp p
  from *(1,5) have d: max-depth-sort-p pn ≤ max-depth-sort-p ?p
    by (auto, auto intro!: max-depth-sort-le)
  from *(1,5) have m: max-dupl-p pn ≤ max-dupl-p ?p
    by (auto, auto intro!: max-dupl-mono)
  have n: num-syms-p pn < num-syms-p ?p
  proof (cases eqc)
    case add
    from *(1,5) add
    show ?thesis by auto (auto intro!: num-syms-subset)
  next
  case empty
  with *(3) obtain eq2 mp2 where mp': mp' = add-mset eq2 mp2 by (cases
mp', auto)
  from *(1,4) mp' have eq2: eq2 ≠ {#}
    unfolding finite-constructor-form-pat-def finite-constructor-form-mp-def by
auto
  from *(1,5) mp' eq2
  show ?thesis by auto (cases eq2, auto intro!: num-syms-subset)
  qed
  from d n m show ?case by (intro measure-p-num-syms, auto)
qed auto

```

snd = Simplified Non-Deterministic

```

inductive-set snd-step :: ('f,'s)simple-pat-problem-ms rel (⟶snd)
  where snd-step: p ⇒ss Q ⇒ finite-constructor-form-pat (mset3 p) ⇒ q ∈#
Q ⇒ (p,q) ∈ ⇒snd

```

```

lemma snd-step-measure: assumes (p,q) ∈ ⇒snd
  shows measure-p p > measure-p q
  using assms
proof (cases)
  case (snd-step Q)
  thus ?thesis using spp-step-complexity-and-termination[of p Q q] by auto
qed

```

lemma *snd-bound-steps*: **assumes** $(p,q) \in \Rightarrow_{snd} \widetilde{\sim} n$
shows $measure-p\ q + n \leq measure-p\ p$
using *steps-bound*[of $\Rightarrow_{snd}\ measure-p\ p\ q\ n$, *OF snd-step-measure assms*]
by *auto*

lemma *snd-steps-bound*: **assumes** $steps: (p,q) \in \Rightarrow_{snd} \widetilde{\sim} n$
and *finite-constructor-form-pat* (*mset3* p)
shows $n \leq card\ (dom\ C) * size\ p * (num-syms-p\ p + 1) * (m + 1)$
proof –
from *snd-bound-steps*[*OF steps*]
have $n \leq measure-p\ p$ **by** *auto*
also have $\dots \leq card\ (dom\ C) * size\ p * (num-syms-p\ p + 1) * (m + 1)$
by (*rule measure-p-bound*[*OF assms*(2)])
finally show *?thesis* .
qed

lemma *SN-snd-step*: $SN \Rightarrow_{snd}$
proof (*rule SN-subset*[*OF SN-inv-image*[*OF SN-nat-gt*]])
show $\Rightarrow_{snd} \subseteq inv-image\ \{(a, b). b < a\}$ **using** *snd-step-measure* **by**
auto
qed

lemma *snd-step-size*: **assumes** $(p,q) \in \Rightarrow_{snd}$
shows $size\ p \geq size\ q$
using *assms*
proof (*cases*)
case (*snd-step* Q)
thus *?thesis* **using** *spp-step-ms-size*[of $p\ Q\ q$] **by** *auto*
qed

lemma *snd-step-finite-constructor-form-pat*: **assumes** $(p,q) \in \Rightarrow_{snd}$
shows *finite-constructor-form-pat* (*mset3* q)
using *assms*
proof (*cases*)
case (*snd-step* Q)
thus *?thesis* **using** *spp-step-ms*[of $p\ Q$] **by** *auto*
qed

lemma *snd-step-completeness*: **assumes** $p \notin NF \Rightarrow_{snd}$
shows *simple-pat-complete* $C\ SS\ (mset3\ p) \longleftrightarrow (\forall\ q. (p,q) \in \Rightarrow_{snd} \longrightarrow sim-
ple-pat-complete\ C\ SS\ (mset3\ q))$
(is *?left* = *?right*)
proof –
from *assms* **obtain** q **where** $(p,q) \in \Rightarrow_{snd}$ **by** *auto*
then obtain Q **where** $p \Rightarrow_{ss}\ Q$ **and** *fin*: *finite-constructor-form-pat* (*mset3* p)
by *cases auto*
from *spp-step-ms*[*OF this*] *snd-step*[*OF this*]
have *oneDir*: *?right* \implies *?left* **by** *auto*
{

```

assume  $\neg$  ?right
then obtain  $q$  where  $(p,q) \in \Rightarrow_{snd}$  and not:  $\neg$  simple-pat-complete  $C$   $SS$ 
(mset3  $q$ ) by auto
from this(1) obtain  $Q$  where  $p \Rightarrow_{ss} Q$  and  $q \in \# Q$  by cases auto
from spp-step-ms[OF this(1) fin] this(2) not
have  $\neg$  ?left by auto
}
with oneDir show ?thesis by auto
qed

```

```

lemma snd-steps: assumes  $(p,q) \in (\Rightarrow_{snd})^*$ 
shows  $size\ p \geq size\ q$ 
simple-pat-complete  $C$   $SS$  (mset3  $p$ )  $\implies$  simple-pat-complete  $C$   $SS$  (mset3  $q$ )
finite-constructor-form-pat (mset3  $p$ )  $\implies$  finite-constructor-form-pat (mset3  $q$ )
using assms
proof (atomize(full), induct, force)
case (step  $q$   $r$ )
from step(2) have  $q \notin NF \Rightarrow_{snd}$  by auto
from snd-step-completeness[OF this]
snd-step-finite-constructor-form-pat[OF step(2)]
snd-step-size[OF step(2)]
step(2-3)
show ?case by auto
qed

```

```

lemma snd-steps-complete: assumes  $\neg$  simple-pat-complete  $C$   $SS$  (mset3  $p$ )
shows  $\exists q. (p,q) \in \Rightarrow_{snd}^! \wedge \neg$  simple-pat-complete  $C$   $SS$  (mset3  $q$ )
using assms
proof (induct  $p$  rule: SN-induct[OF SN-snd-step])
case (1  $p$ )
show ?case
proof (cases  $p \in NF \Rightarrow_{snd}$ )
case True
thus ?thesis using 1 by auto
next
case False
from snd-step-completeness[OF False] 1 obtain  $q$  where step:  $(p,q) \in \Rightarrow_{snd}$ 
and  $q: \neg$  simple-pat-complete  $C$   $SS$  (mset3  $q$ ) by auto
from 1(1)[OF this] step show ?thesis by blast
qed
qed

```

```

lemma simple-pat-complete-via-snd-step-NFs: simple-pat-complete  $C$   $SS$  (mset3  $p$ )
 $\longleftrightarrow$ 
 $(\forall q. (p,q) \in \Rightarrow_{snd}^! \longrightarrow$  simple-pat-complete  $C$   $SS$  (mset3  $q$ ))
using snd-steps-complete[of  $p$ ] snd-steps(2)[of  $p$ ] by blast

```

```

lemma snd-steps-NF-fvf-small-sort: assumes finite-constructor-form-pat (mset3
 $p$ )

```

and $(p, q) \in \Rightarrow_{snd}^!$
shows $\forall mp \in \# q. \forall eqc \in \# mp. \forall t \in \# eqc. \exists x \iota. t = \text{Var } (x, \iota) \wedge \text{card-of-sort } C \iota \leq \text{size } p$
proof –
from *snd-steps*[of p q] *assms*
have *fin*: *finite-constructor-form-pat* (*mset3* q) **and**
size: *size* $p \geq \text{size } q$ **by** *blast+*
from *assms* **have** $q \in NF (\Rightarrow_{snd})$ **by** *auto*
hence $\neg (\exists Q. q \Rightarrow_{ss} Q \wedge Q \neq \{\#\})$ **using** *snd-step*[*OF* - *fin*] **by** *blast*
from *NF-spp-step-fvf-with-small-card*[*OF fin this*] *size* **show** *?thesis* **by** *fastforce*
qed

Major soundness properties of non-deterministic algorithm to transform FCF into FVF

lemmas *snd-step-combined* =
snd-steps-bound — complexity
SN-snd-step — termination
snd-steps-NF-fvf-small-sort — normal forms are in finite variable form with small sorts
simple-pat-complete-via-snd-step-NFs — equivalence: complete iff all normal forms are complete

inductive-set *spp-det-step-ms* :: $(f, 's)\text{simple-pat-problem-ms multiset rel } (\Leftarrow_{ss}^!)$
where *spp-non-det-step*: $p \Rightarrow_{ss} P' \Longrightarrow \text{finite-constructor-form-pat } (mset3\ p) \Longrightarrow (\text{add-mset } p\ P, P' + P) \in \Rightarrow_{ss}$
| *spp-non-det-fail*: $P \neq \{\#\} \Longrightarrow (\text{add-mset } \{\#\}\ P, \{\#\{\#\}\#\}) \in \Rightarrow_{ss}$
| *spp-fvf-succ*: $(\bigwedge t. t \in \# \sum_{\#} (\text{image-mset } \sum_{\#} p) \Longrightarrow \text{is-Var } t) \Longrightarrow \text{simple-pat-complete } C\ SS\ (mset3\ p) \Longrightarrow (\text{add-mset } p\ P, P) \in \Rightarrow_{ss}$
| *spp-fvf-fail*: $(\bigwedge t. t \in \# \sum_{\#} (\text{image-mset } \sum_{\#} p) \Longrightarrow \text{is-Var } t) \Longrightarrow \neg \text{simple-pat-complete } C\ SS\ (mset3\ p) \Longrightarrow \text{finite-constructor-form-pat } (mset3\ p) \Longrightarrow p \neq \{\#\} \Longrightarrow (\text{add-mset } p\ P, \{\#\{\#\}\#\}) \in \Rightarrow_{ss}$

definition *spp-det-prob* :: $(f, 's)\text{simple-pat-problem-ms multiset} \Rightarrow \text{bool}$ **where**
spp-det-prob $P = (\forall p \in \# P. \text{finite-constructor-form-pat } (mset3\ p))$

definition *spp-pat-complete* :: $(f, 's)\text{simple-pat-problem-ms multiset} \Rightarrow \text{bool}$ **where**
spp-pat-complete $P = (\forall p \in \# P. \text{simple-pat-complete } C\ SS\ (mset3\ p))$

lemma *spp-det-prob-add*[*simp*]: *spp-det-prob* (*add-mset* p P) =
(*finite-constructor-form-pat* (*mset3* p) \wedge *spp-det-prob* P)
unfolding *spp-det-prob-def* **by** *simp*

lemma *spp-det-prob-plus*[*simp*]: *spp-det-prob* ($P + P'$) =
(*spp-det-prob* $P \wedge$ *spp-det-prob* P')

unfolding *spp-det-prob-def* **by** *auto*

lemma *spp-det-prob-empty[simp]*: *spp-det-prob* {#}
by (*simp add: spp-det-prob-def*)

lemma *spp-det-step-ms*: $(P, P') \in \Rightarrow_{ss} \implies$
spp-pat-complete $P = \text{spp-pat-complete } P' \wedge (\text{spp-det-prob } P \longrightarrow \text{spp-det-prob } P')$
proof (*induct rule: spp-det-step-ms.induct*)
case *: (*spp-non-det-step* p $P' P$)
from *spp-step-ms*[*OF* *(1), *folded spp-det-prob-def*] *(2)
show ?*case* **by** (*auto simp: spp-pat-complete-def*)
next
case *: (*spp-non-det-fail* P)
show ?*case* **using** *detect-unsat-spp* **by** (*simp add: finite-constructor-form-pat-def*
spp-pat-complete-def)
next
case (*spp-fvf-succ* p P)
thus ?*case* **by** (*auto simp: spp-pat-complete-def*)
next
case (*spp-fvf-fail* p P)
thus ?*case* **using** *detect-unsat-spp* **by** (*simp add: finite-constructor-form-pat-def*
spp-pat-complete-def)
qed

lemma *spp-det-steps-ms*: $(P, P') \in \Rightarrow_{ss}^* \implies$
spp-pat-complete $P = \text{spp-pat-complete } P' \wedge (\text{spp-det-prob } P \longrightarrow \text{spp-det-prob } P')$
by (*induct rule: rtrancl-induct, insert spp-det-step-ms, auto*)

lemma *SN-spp-det-step*: $SN \Rightarrow_{ss}$
proof (*rule SN-subset*)
show $\Rightarrow_{ss} \subseteq (\text{mult } (\text{measure measure-}p))^{-1}$
proof (*standard, simp, goal-cases*)
case (1 $P P'$)
thus ?*case*
proof (*induct rule: spp-det-step-ms.induct*)
case *: (*spp-non-det-step* p $P' P$)
from *spp-step-complexity-and-termination*[*OF* *(1-2)] **show** ?*case*
by (*simp add: add-many-mult*)
next
case *: (*spp-non-det-fail* P)
then obtain p P' **where** $P: P = \text{add-mset } p$ P' **by** (*cases P, auto*)
show ?*case* **unfolding** P **by** (*simp add: subset-implies-mult*)
next
case (*spp-fvf-succ* p P)
show ?*case* **by** (*simp add: subset-implies-mult*)
next
case *: (*spp-fvf-fail* p P)
then obtain mp p' **where** $p: p = \text{add-mset } mp$ p' **by** (*cases p, auto*)
with *(2) **have** $mp \neq \{\#\}$ **unfolding** *simple-pat-complete-def simple-match-complete-wrt-def*

```

by auto
  then obtain eq mp' where mp: mp = add-mset eq mp' by (cases mp, auto)
  with *(3,4) have eq: eq ≠ {#}
  unfolding finite-constructor-form-pat-def finite-constructor-form-mp-def mp
p
  by auto
  have p: measure-p p > 0 unfolding measure-p-def p num-syms-p-def num-syms-smp-def
mp
  using eq by (cases eq, auto)
  have e: measure-p {#} = 0 unfolding measure-p-def p num-syms-p-def
max-depth-sort-p-def
  max-dupl-p-def by auto
  from p e have ({#}, p) ∈ measure measure-p by auto
  thus ?case
  by (metis add-cancel-left-left add-mset-eq-single empty-not-add-mset multi-member-split
one-step-implies-mult union-single-eq-member)
qed
qed
show SN ((mult (measure measure-p))-1)
  by (rule wf-imp-SN, simp add: wf-mult[OF wf-measure])
qed

lemma NF-spp-det-step: assumes spp-det-prob P
  and P ∈ NF ≅ss
shows P = {#} ∨ P = {#{#}#}
proof (rule ccontr)
  assume contr: ¬ ?thesis
  from assms(2) have NF: (P,P') ∈ ≅ss ⇒ False for P' by auto
  from contr obtain p P2 where P: P = add-mset p P2 and disj: p ≠ {#} ∨
P2 ≠ {#} by (cases P, auto)
  from assms[unfolded P] have fin: finite-constructor-form-pat (mset3 p) by auto

  show False
  proof (cases p = {#})
    case True
    with disj have P2: P2 ≠ {#} by auto
    show False
    apply (rule NF)
    apply (unfold P True)
    by (rule spp-non-det-fail[OF P2])
  next
    case False
    have ‡P. p ⇒ss P ∧ P ≠ {#} using NF[unfolded P, OF spp-non-det-step[OF
- fin]] by auto
    note fvf = normal-form-spp-step-fvf[OF fin this]
    show ?thesis
    proof (cases simple-pat-complete C SS (mset3 p))
      case True
      show False

```

```

      by (rule NF, unfold P, rule spp-fvf-succ[OF - True], insert fvf, auto)
    next
      case unsat: False
      show False
      by (rule NF, unfold P, rule spp-fvf-fail[OF - unsat fin False], insert fvf,
    auto)
  qed
qed
qed

```

lemma *decision-procedure-spp-det*:

```

  assumes valid-input: spp-det-prob P and NF: (P,Q) ∈ ⇔ss!
  shows Q = {#} ∧ spp-pat-complete P — either the result is and input P is
  complete
  ∨ Q = {#{#}#} ∧ ¬ spp-pat-complete P — or the result = bot and P is not
  complete
  proof —
    from NF have steps: (P,Q) ∈ ⇔ss* and Q: Q ∈ NF ⇔ss by auto
    from spp-det-steps-ms[OF steps] valid-input
    have equiv: spp-pat-complete P = spp-pat-complete Q and valid-out: spp-det-prob
    Q by auto
    from NF-spp-det-step[OF valid-out Q]
    show ?thesis
    proof
      assume Q = {#}
      thus ?thesis unfolding equiv by (simp add: spp-pat-complete-def)
    next
      assume Q = {#{#}#}
      thus ?thesis unfolding equiv using detect-unsat-spp by (simp add: spp-pat-complete-def)
    qed
  qed

```

A combined complexity approximation

Conversion from pattern problems in finite constructor form to their simplified representation, performed on multiset-representation

definition *fcf-mp-to-smp* :: ('f,'v,'s)match-problem-mset ⇒ ('f,'s)simple-match-problem-ms
where

```

  fcf-mp-to-smp mp = (let xs = mset-set ((the-Var o snd) ` set-mset mp)
    in image-mset (λ x. image-mset fst (filter-mset (λ (t,l). l = Var x) mp)) xs)

```

lemma *fcf-mp-to-smp*: **assumes** *fcf*: finite-constr-form-mp C (set-mset mp)

and *no-fail*: ¬ match-fail mp

and *wf-match*: wf-match (set-mset mp)

shows finite-constructor-form-mp (mset2 (fcf-mp-to-smp mp))

unfolding finite-constructor-form-mp-def

proof

fix seqc

assume seqc ∈ mset2 (fcf-mp-to-smp mp)

```

from this[unfolded fcf-mp-to-smp-def Let-def, simplified]
obtain  $x$   $tx$   $eqc$  where  $tx: tx \in \# mp$  and  $x: x = the-Var (snd tx)$ 
  and  $eqc: eqc = image-mset fst \{\#(t, l) \in \# mp. l = Var x\}$ 
  and  $seqc: seqc = set-mset eqc$  by force
from fcf[unfolded finite-constr-form-mp-def, rule-format, of fst tx snd tx]  $tx$   $x$ 
obtain  $t$   $\iota$  where  $tx: (t, Var x) \in \# mp$  and  $fin: finite-sort C \iota$  and  $t: t : \iota$  in
 $\mathcal{T}(C, \mathcal{V})$ 
  by (cases tx, auto)
from  $tx$   $eqc$   $seqc$  have  $seqc1: seqc \neq \{\}$  by auto
{
  fix  $t'$ 
  assume  $t' \in seqc$ 
  from this[unfolded seqc eqc] have  $t': (t', Var x) \in \# mp$  by auto
  have  $\mathcal{T}(C, \mathcal{V}) t' = \mathcal{T}(C, \mathcal{V}) t$ 
  proof (rule ccontr)
    let  $?p1 = (t, Var x)$ 
    let  $?p2 = (t', Var x)$ 
    define  $mp'$  where  $mp' = mp - \{\# ?p1, ?p2 \# \}$ 
    assume  $\neg ?thesis$ 
    hence  $?p1 \neq ?p2$  by auto
    with  $tx$   $t'$  have  $mp = add-mset ?p1 (add-mset ?p2 mp^)$ 
      unfolding  $mp'$ -def using multi-member-split by fastforce
    from match-clash-sort[of t t' x mp', folded this]  $\langle \neg ?thesis \rangle$  no-fail
    show False by auto
  qed
  with  $t$  have  $t'i: t' : \iota$  in  $\mathcal{T}(C, \mathcal{V})$  unfolding hastype-def by auto
  from wf-match[unfolded wf-match-def tvars-match-def]  $t'$  have  $vars t' \subseteq SS$ 
by force
  with  $t'i$  have  $t' : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid SS)$ 
    by (meson hastype-in-Term-mono-right hastype-in-Term-restrict-vars restrict-map-mono-right)
  }
  with  $fin$   $seqc1$ 
  show  $seqc \neq \{\}$   $\wedge (\exists \iota. finite-sort C \iota \wedge (\forall t \in seqc. t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid SS)))$ 
  by auto
qed

```

definition $fcf-pat-to-spat :: ('f, 'v, 's)pat-problem-mset \Rightarrow ('f, 's)simple-pat-problem-ms$
where
 $fcf-pat-to-spat = image-mset fcf-mp-to-smp$

lemma $fcf-pat-to-spat$: **assumes** $fcf: finite-constr-form-pat C (pat-mset p)$
and $NF: p \in NF (\Rightarrow_{nd})$
and $wf-pat: wf-pat (pat-mset p)$
shows $finite-constructor-form-pat (mset3 (fcf-pat-to-spat p))$
unfolding $finite-constructor-form-pat-def$
proof
fix $ssmp$
assume $ssmp \in mset3 (fcf-pat-to-spat p)$

```

then obtain smp where smp ∈# fcf-pat-to-spat p and ssmp: ssmp = mset2 smp
by auto
from this[unfolded fcf-pat-to-spat-def] obtain mp where mp: mp ∈# p and
  smp: smp = fcf-mp-to-smp mp by auto
with ssmp have ssmp: ssmp = mset2 (fcf-mp-to-smp mp) by auto
show finite-constructor-form-mp ssmp unfolding ssmp
proof (rule fcf-mp-to-smp)
  from fcf[unfolded finite-constr-form-pat-def] mp
  show finite-constr-form-mp C (mp-mset mp) by auto
  from wf-pat[unfolded wf-pat-def] mp
  show wf-match (mp-mset mp) by auto
  show ¬ match-fail mp
proof
  from mp obtain p' where p: p = add-mset mp p' by (metis mset-add)
  assume match-fail mp
  from pat-remove-mp[OF this, of p', folded p]
  have p ⇒m {#p'#} .
  hence (p,p') ∈ ⇒nd by (intro pp-nd-step-mset.intros, auto)
  with NF show False by auto
qed
qed
qed

```

```

lemma wf-pat-nd-step: (p,q) ∈ ⇒nd ⇒ wf-pat (pat-mset p) ⇒ wf-pat (pat-mset
q)
by (smt (verit, ccfv-SIG) comp-def image-iff p-step-mset-imp-set pp-nd-step-mset.cases
pp-step-wf)

```

```

lemma sum-smp-fcf-mp-to-smp-is-image-fst:
assumes finite-constr-form-mp C (mp-mset mp)
shows sum-mset (fcf-mp-to-smp mp) = image-mset fst mp
using assms
proof (induct size mp arbitrary: mp rule: less-induct)
case (less mp)
show ?case
proof (cases mp)
  case empty
  thus ?thesis by (auto simp: fcf-mp-to-smp-def)
next
case (add tx mp')
from less(2)[unfolded finite-constr-form-mp-def add]
obtain t x where tx: tx = (t, Var x) by (cases tx; cases snd tx; auto)
define mp1 where mp1 = filter-mset (λ ty. snd ty = Var x) mp
from tx add have tx: (t, Var x) ∈# mp1 by (auto simp: mp1-def)
define mp2 where mp2 = filter-mset (Not o (λ ty. snd ty = Var x)) mp
define ys where ys = mset-set {the-Var (snd y) | y ∈ mp-mset mp2}
from tx have x: x ∈ {the-Var (snd x) | x ∈ mp-mset mp1 ∪ mp-mset mp2}
by force
from less(2) have fcf: finite-constr-form-mp C (mp-mset mp2)

```

unfolding *finite-constr-form-mp-def mp2-def* **by** *auto*
have *mp*: $mp = mp1 + mp2$ **unfolding** *mp1-def mp2-def* **by** *auto*
have *x2*: $x \notin ((the-Var \circ snd) \text{ ' } mp\text{-mset } mp2)$ **unfolding** *mp2-def*
using *less(2)[unfolded finite-constr-form-mp-def]*
by *auto*
hence *xy*: $x \notin \# \text{ } ys$ **unfolding** *ys-def* **by** *auto*
have $((the-Var \circ snd) \text{ ' } mp\text{-mset } mp =$
 $\text{insert } x \text{ } (((the-Var \circ snd) \text{ ' } mp\text{-mset } mp2)))$
using *x* **unfolding** *mp* **by** $(auto \text{ simp: } mp1\text{-def})$
from *arg-cong[OF this, of mset-set]*
have *xs*: $mset\text{-set } (((the-Var \circ snd) \text{ ' } mp\text{-mset } mp) = add\text{-mset } x \text{ } (mset\text{-set } (((the-Var \circ snd) \text{ ' } mp\text{-mset } mp2)))$
using *x2* **by** *auto*
have *fcf-mp-to-smp mp* = $add\text{-mset } (image\text{-mset } fst \text{ } mp1)$
 $\{\#image\text{-mset } fst \text{ } \{\#(t, l) \in \# \text{ } mp. l = Var \text{ } y\#\}. y \in \# \text{ } ys\#\}$
unfolding *fcf-mp-to-smp-def xs Let-def ys-def*
by $(auto \text{ simp } add: mp1\text{-def } intro!: arg\text{-cong}[of \text{ - } image\text{-mset } fst]) \text{ } (induct \text{ } mp, \text{ } auto)$
also have $\{\#image\text{-mset } fst \text{ } \{\#(t, l) \in \# \text{ } mp. l = Var \text{ } y\#\}. y \in \# \text{ } ys\#\} =$
 $\{\#image\text{-mset } fst \text{ } \{\#(t, l) \in \# \text{ } mp2. l = Var \text{ } y\#\}. y \in \# \text{ } ys\#\}$
unfolding *mp* **using** *xy*
proof $(induct \text{ } ys)$
case $(add \text{ } y \text{ } ys)$
hence $x \neq y$ **by** *auto*
hence $\{\#(t, l) \in \# \text{ } mp1. l = Var \text{ } y\#\} = \{\#\}$ **unfolding** *mp1-def* **by** $(induct \text{ } mp, \text{ } auto)$
with *add show ?case* **by** *auto*
qed *auto*
also have $\dots = fcf\text{-mp-to-smp } mp2$ **unfolding** *fcf-mp-to-smp-def Let-def ys-def*
by *auto*
finally have $sum\text{-mset } (fcf\text{-mp-to-smp } mp) =$
 $image\text{-mset } fst \text{ } mp1 + sum\text{-mset } (fcf\text{-mp-to-smp } mp2)$
by $(auto \text{ simp: } num\text{-syms-smp-def})$
also have $sum\text{-mset } (fcf\text{-mp-to-smp } mp2) = image\text{-mset } fst \text{ } mp2$ **using** *less(1)[OF - fcf]*
unfolding *mp* **using** *tx* **by** $(cases \text{ } mp1, \text{ } auto)$
finally show *?thesis* **unfolding** *mp* **by** *auto*
qed
qed

lemma *num-syms-smp-fcf-mp-to-smp-is-meas-tsymbols:*

assumes *finite-constr-form-mp C (mp-mset mp)*

shows $num\text{-syms-smp } (fcf\text{-mp-to-smp } mp) = num\text{-tsyms-mp } mp$

proof –

let *?mp* = $fcf\text{-mp-to-smp } mp$

have $num\text{-syms-smp } ?mp = (\sum x \in \# \text{ } mp. num\text{-syms } (fst \text{ } x))$

unfolding *num-syms-smp-def sum-smp-fcf-mp-to-smp-is-image-fst[OF assms]*
image-mset.compositionality o-def

by *simp*

also have $\dots = \text{num-tsyms-mp } mp$
unfolding $\text{num-tsyms-mp-def o-def}$ **by** simp
finally show $?thesis$.
qed

lemma $\text{num-syms-p-fcf-pat-to-spat-is-meas-tsymbols}$:
assumes $\text{finite-constr-form-pat } C$ ($\text{pat-mset } p$)
shows $\text{num-syms-p (fcf-pat-to-spat } p) = \text{meas-tsymbols } p$
using $\text{num-syms-smp-fcf-mp-to-smp-is-meas-tsymbols } \text{assms}$
unfolding $\text{num-syms-p-def meas-tsymbols-def finite-constr-form-pat-def fcf-pat-to-spat-def}$
unfolding $\text{image-mset.compositionality o-def}$
by ($\text{metis image-eqI image-mset-cong}$)

lemma $\text{measure-pat-poly-to-measure-p}$:
assumes $\text{finite-constr-form-pat } C$ ($\text{pat-mset } p$)
and $\text{finite-constructor-form-pat (mset3 (fcf-pat-to-spat } p))}$
and c : $\text{card (dom } C) * \text{size } p \leq c$
shows $\text{measure-p (fcf-pat-to-spat } p) \leq \text{measure-pat-poly } c$ p
proof –
show $?thesis$
unfolding $\text{measure-p-def measure-pat-poly-def}$
unfolding $\text{num-syms-p-fcf-pat-to-spat-is-meas-tsymbols[OF assms(1)]}$
proof ($\text{intro add-mono le-refl mult-mono}$)
from $\text{max-depth-sort-p-le-card-size[OF assms(2)] } c$
show $\text{max-depth-sort-p (fcf-pat-to-spat } p) \leq c + \text{meas-diff } p$
unfolding $\text{fcf-pat-to-spat-def}$ **by** simp
have $\text{max-dupl-p (fcf-pat-to-spat } p) = \text{meas-dupl } p$
unfolding $\text{meas-dupl-def max-dupl-p-def fcf-pat-to-spat-def image-mset.compositionality}$
 o-def
proof ($\text{intro arg-cong[of - - sum-mset] image-mset-cong}$)
fix mp
assume $mp \in \# p$
with $\text{assms[unfolded finite-constr-form-pat-def]}$
have $\text{fcf: finite-constr-form-mp } C$ ($\text{mp-mset } mp$) **by** auto
have $\bigcup (\bigcup ((\cdot) \text{ vars ' mset2 (fcf-mp-to-smp } mp)))$
 $= \bigcup (\text{vars ' (set-mset } (\sum \# (\text{fcf-mp-to-smp } mp)))$ **by** auto
also have $\dots = \bigcup (\text{vars ' fst ' set-mset } mp)$ **unfolding**
 $\text{sum-smp-fcf-mp-to-smp-is-image-fst[OF fcf]}$ **by** auto
also have $\dots = \text{tvars-match (mp-mset } mp)$
unfolding $\text{tvars-match-def o-def}$ **by** auto
finally have $\text{id1: } \bigcup (\bigcup ((\cdot) \text{ vars ' mset2 (fcf-mp-to-smp } mp))) = \text{tvars-match}$
 $(\text{mp-mset } mp)$.
show $\text{max-dupl-smp (fcf-mp-to-smp } mp) = \text{max-dupl-mp } mp$
unfolding $\text{max-dupl-smp-def max-dupl-mp-def id1}$
unfolding $\text{image-mset.compositionality o-def}$
unfolding $\text{sum-smp-fcf-mp-to-smp-is-image-fst[OF fcf]}$
proof ($\text{intro arg-cong[of - - Max] arg-cong[of - - insert 0] image-cong refl}$)
fix x

```

    show  $(\sum t \in \# \text{image-mset } \text{fst } mp. \text{count } (\text{syms-term } t) (\text{Inl } x)) = (\sum tl \in \# mp. \text{count } (\text{syms-term } (\text{fst } tl)) (\text{Inl } x))$ 
    by (induct mp, auto)
  qed
  qed
  thus  $\text{max-dupl-}p (\text{fcf-pat-to-spat } p) \leq \text{meas-dupl } p$  by simp
  qed auto
  qed

```

```

theorem complexity-combined: fixes  $p :: ('f, 'v, 's)\text{pat-problem-mset}$ 
  assumes impr: improved_infinite (UNIV :: 'v set)
  and wf: wf_pat (pat_mset p)
  and phase1:  $(p, p1) \in \Rightarrow_{nd} \sim n1$   $(p, p1) \in \Rightarrow_{nd}^!$ 
  and translation:  $p1' = \text{fcf-pat-to-spat } p1$ 
  and phase2:  $(p1', p2) \in \Rightarrow_{snd} \sim n2$ 
  shows  $n1 + n2 + \text{num-syms-p } p2 \leq (\text{card } (\text{dom } C) * \text{size } p + \text{num-syms-pat } p) * (\text{num-syms-pat } p * m + 2)$ 
  size  $p2 \leq \text{size } p$ 
   $p2 \in NF \Rightarrow_{snd} \Rightarrow mp \in \# p2 \Rightarrow \text{eqc} \in \# mp \Rightarrow t \in \# \text{eqc} \Rightarrow (x, t) \in \text{vars } t \Rightarrow \text{card-of-sort } C \iota \leq \text{size } p$ 
  proof -
  define c where  $c = \text{card } (\text{dom } C) * \text{size } p$ 
  note nd = nd_step_improved[OF impr wf]
  from nd(2)[OF phase1(1), of c]
  have n1:  $\text{measure-pat-poly } c p1 + n1 \leq (c + \text{num-syms-pat } p) * (\text{num-syms-pat } p * m + 2)$  .
  from phase1(2) have NF:  $p1 \in NF (\Rightarrow_{nd})$  by auto
  from phase1 have star:  $(p, p1) \in \Rightarrow_{nd}^*$  by auto
  from nd-steps-le-size[OF star]
  have pp1:  $\text{size } p1 \leq \text{size } p$  and c:  $\text{card } (\text{dom } C) * \text{size } p1 \leq c$  unfolding c-def
  by auto
  from pp1 have p1'p:  $\text{size } p1' \leq \text{size } p$  unfolding translation fcf-pat-to-spat-def
  by auto
  from nd(3)[OF phase1(2)]
  have fcf: finite-constr-form-pat C (pat_mset p1) by auto
  from star wf have wf_pat (pat_mset p1) using wf_pat-nd-step by (rule rtrancl-induct)
  from fcf-pat-to-spat[OF fcf NF this, folded translation]
  have fcf': finite-constructor-form-pat (mset3 p1') .
  from measure-pat-poly-to-measure-p[OF fcf, folded translation, OF fcf' c] n1
  have measure-p  $p1' + n1 \leq (c + \text{num-syms-pat } p) * (\text{num-syms-pat } p * m + 2)$ 

  by auto
  with snd-bound-steps[OF phase2(1)]
  have measure-p  $p2 + n2 + n1 \leq (c + \text{num-syms-pat } p) * (\text{num-syms-pat } p * m + 2)$  by auto
  moreover have num-syms-p  $p2 \leq \text{measure-p } p2$  unfolding measure-p-def by auto
  ultimately show  $n1 + n2 + \text{num-syms-p } p2 \leq (c + \text{num-syms-pat } p) *$ 

```

```

(num-syms-pat p * m + 2) by auto
  from phase2 have (p1',p2) ∈ ⇒snd* by (rule relpow-imp-rtrancl)
  with fcf' have size p2 ≤ size p1' by (metis snd-steps(1))
  with p1'p show size p2 ≤ size p by simp

  assume NF: p2 ∈ NF ⇒snd and mp: mp ∈# p2 and e: eqc ∈# mp and t: t
  ∈# eqc and x: (x,ι) ∈ vars t
  from NF phase2 have (p1',p2) ∈ ⇒snd!
  by (meson normalizability-I relpow-imp-rtrancl)
  from snd-steps-NF-fvf-small-sort[OF fcf' this, rule-format, OF mp e t] x
  have card-of-sort C ι ≤ size p1' by auto
  also have ... ≤ size p by fact
  finally show card-of-sort C ι ≤ size p .
qed
end

end
theory FCF-List
  imports
    FCF-Multiset
    Singleton-List
    Finite-IDL-Solver-Interface
    Pattern-Completeness-List
begin

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 { $\alpha v \mid v. (Var v, Var x) \in set (match-of-var-form-list mpv)$ } (is
UNIQ ?vs)
  proof (cases x  $\in$  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)  $\in$  set mpv by force
        with r have uniq: UNIQ ( $\alpha$  ' 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  $\notin$  fst ' set bef  $\cup$  fst ' set aft by auto
        hence  $\alpha$  ' 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  $\in$  pat-list pp and uniq:  $\forall x. UNIQ \{\alpha v \mid v. (Var v, Var x) \in mp\}$ 
  from this[unfolded pp pat-list-def pat-of-var-form-list-def]
  obtain mpv where mem: mpv  $\in$  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  $\exists mpv \in set fvf. \forall (x, vs) \in set mpv. UNIQ (\alpha ' set vs)$ 
  proof (intro beXI[OF - mem], safe)
    fix x vs
    assume (x,vs)  $\in$  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  $\notin$  fst ' set bef  $\cup$  fst ' set aft by auto
    from uniq[rule-format, of x]
    have UNIQ { $\alpha v \mid v. (Var v, Var x) \in mp$ } .
    also have { $\alpha v \mid v. (Var v, Var x) \in mp$ } =  $\alpha$  ' set vs
      unfolding mp match-of-var-form-list-def mpv using x by force
    finally show UNIQ ( $\alpha$  ' 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 o map fst)
    and cnf: cnf = map (map snd) fvf
  shows pat-complete C (pat-list pp)  $\longleftrightarrow$ 
    ( $\forall \alpha. (\forall v \in \text{set } (\text{concat } (\text{concat } \text{cnf})). \alpha v < \text{card-of-sort } C (\text{snd } v)) \longrightarrow$ 
      ( $\exists c \in \text{set } \text{cnf}. \forall vs \in \text{set } c. \text{UNIQ } (\alpha \text{ ' set } vs)))$ )
  unfolding finite-var-form-pat-pat-complete-list[OF fvf pp dist] cnf[symmetric]
proof (intro all-cong1 arg-cong[of - -  $\lambda x. x \longrightarrow -$ ] 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

```

```

fun zip-lists :: nat  $\Rightarrow$  'a list list  $\Rightarrow$  'a list list where
  zip-lists n [] = replicate n []
| zip-lists n (xs # xss) = map2 (#) xs (zip-lists n xss)

```

```

lemma zip-lists: assumes  $\bigwedge xs. xs \in \text{set } xss \implies \text{length } xs = n$ 
  shows zip-lists n xss = map ( $\lambda i. \text{map } (\lambda xs. xs ! i) xss$ ) [0.. $n$ ]
  using assms
proof (induct xss)
  case Nil
  then show ?case by (auto intro: nth-equalityI)
next
  case (Cons xs xss)
  hence IH: zip-lists n xss = map ( $\lambda i. \text{map } (\lambda xs. xs ! i) xss$ ) [0.. $n$ ]
    and len: length xs = n by auto
  show ?case unfolding zip-lists.simps IH map-map using len
    by (intro nth-equalityI, auto)
qed

```

```

fun length-gt-1 where
  length-gt-1 (x # y # xs) = True
| length-gt-1 - = False

```

```

lemma length-gt-1[simp]: length-gt-1 xs = (length xs > 1)
  by (cases xs rule: length-gt-1.cases, auto)

```

```

definition uniq-list :: 'a list  $\Rightarrow$  bool where
  uniq-list xs = (xs = []  $\vee$  is-singleton-list xs)

```

```

lemma uniq-list[simp]: uniq-list xs  $\longleftrightarrow$  UNIQ (set xs)
  unfolding uniq-list-def is-singleton-list2
  by (cases xs; auto simp: Uniq-def)

```

primrec *extract-option* :: ('a ⇒ 'b option) ⇒ 'a list ⇒ ('a list × 'a × 'b × 'a list)option **where**

extract-option f [] = None
| *extract-option* f (x # xs) = (case f x of Some y ⇒ Some ([], x, y, xs)
| None ⇒ map-option (map-prod (Cons x) id) (*extract-option* f xs))

lemma *extract-option-Some*: **assumes** *extract-option* f xs = Some (bef, x, y, aft)

shows xs = bef @ x # aft f x = Some y

using *assms*

proof (*atomize(full)*, *induct xs arbitrary: bef*)

case (Cons u us)

show ?case

by (*cases f u*, *insert Cons*, *auto*)

qed *simp*

lemma *extract-option-None*: *extract-option* f xs = None ⟷ Ball (set xs) (λ x. f x = None)

by (*induct xs*, *auto split: option.splits*)

fun *sort-of* :: ('f × 's, 'v × 's)term ⇒ 's **where**

sort-of (Fun (f,s) ts) = s

| *sort-of* (Var (x,s)) = s

lemma *sort-of-Var* [*simp*]:

⟨*sort-of* (Var x-s) = snd x-s⟩

by (*cases x-s*) *simp*

fun *list-Union* :: 'a set list ⇒ 'a set **where**

list-Union [] = {}

| *list-Union* (x # xs) = x ∪ *list-Union* xs

lemma *list-Union*[*simp*]: *list-Union* xs = ⋃ (set xs)

by (*induct xs*, *auto*)

fun *sorts-of* :: ('f × 's, 'v × 's)term ⇒ 's set **where**

sorts-of (Fun (f,s) ts) = insert s (*list-Union* (map *sorts-of* ts))

| *sorts-of* (Var (x,s)) = {s}

definition *arg-sorts-of* :: ('f × 's, 'v × 's)term ⇒ 's set **where**

arg-sorts-of t = *list-Union* (map *sorts-of* (args t))

fun *remove-sort* :: ('f × 's, 'v)term ⇒ ('f, 'v)term **where**

remove-sort (Var x) = Var x

| *remove-sort* (Fun (f,s) ts) = Fun f (map *remove-sort* ts)

type-synonym ('f, 's)simple-match-problem-list = ('f, nat × 's)term list list

type-synonym (*f*, *s*)*simple-match-problem-slist* = (*f* × *s*, nat × *s*)*term list list*
type-synonym (*f*, *s*)*simple-pat-problem-slist* = (*f* × *s*, nat × *s*)*term list list list*
type-synonym (*f*, *s*)*tagged-simple-pat-problem-slist* = bool × (*f*, *s*)*simple-pat-problem-slist*

definition *search-fun-pp* :: (*f*, *s*)*simple-pat-problem-slist* ⇒ - *option* **where**
search-fun-pp pp = *extract-option* (*extract-option* (*List.extract is-Fun*)) *pp*

lemma *search-fun-pp-None*: **assumes** *search-fun-pp pp* = *None*
shows $t \in \bigcup (\bigcup (\text{set3 } pp)) \implies \text{is-Var } t$
using *assms*[*unfolded search-fun-pp-def*, *unfolded extract-option-None extract-None-iff*]
by *auto*

lemma *search-fun-pp-Some*: **assumes** *search-fun-pp pp* = *Some* (*p1*, *mp*, (*mp1*, *eqc*, (*eqc1*, *t*, *eqc2*), *mp2*), *p2*)

shows $pp = p1 @ mp \# p2$ $mp = mp1 @ eqc \# mp2$ $eqc = eqc1 @ t \# eqc2$
is-Fun t

proof (*atomize(full)*, *goal-cases*)

case 1

from *extract-option-Some*[*OF assms*[*unfolded search-fun-pp-def*]]

have *pp*: $pp = p1 @ mp \# p2$ **and**

Some: *extract-option* (*List.extract is-Fun*) *mp* = *Some* (*mp1*, *eqc*, (*eqc1*, *t*, *eqc2*), *mp2*)

by *auto*

from *extract-option-Some*[*OF Some*]

have *mp*: $mp = mp1 @ eqc \# mp2$ **and**

Some: *List.extract is-Fun* *eqc* = *Some* (*eqc1*, *t*, *eqc2*)

by *auto*

from *List.extract-SomeE*[*OF Some*]

have *eqc*: $eqc = eqc1 @ t \# eqc2$ **and**

t: *is-Fun t*

by *auto*

show *?case* **using** *pp mp eqc t* **by** *auto*

qed

definition *aroot* **where** *aroot* = *map-option* (*map-prod fst id*) *o* *root*

definition *bounds-list* *bnd* *cnf* = (*let* *vars* = *remdups* (*concat* (*concat* *cnf*)))
in *map* ($\lambda v. (v, \text{int } (\text{bnd } v) - 1)$) *vars*)

context *pattern-completeness-context*

begin

fun *add-sort* :: (*f*, nat × *s*)*term* ⇒ (*f* × *s*, nat × *s*)*term* **where**

add-sort (*Var x*) = (*Var x*)

| *add-sort* (*Fun f ts*) = (*let* *ats* = *map* *add-sort* *ts*

in (*Fun* (*f*, *the* (*C* (*f*, *map* *sort-of* *ats*)))) *ats*)

lemma *aroot[simp]*: *aroot* (*add-sort t*) = *root t*

by (*cases t*, *auto simp: aroot-def*)

lemma *remove-add-sort*[simp]: *remove-sort* (*add-sort* *t*) = *t*
 by (*induct* *t*, *auto* *simp*: *o-def* *intro*: *nth-equalityI*)

lemma *is-Var-add-sort*[simp]: *is-Var* (*add-sort* *t*) = *is-Var* *t*
 by (*cases* *t*, *auto*)

lemma *inj-add-sort*[simp]: *inj* *add-sort*
 using *remove-add-sort* by (*metis* *injI*)

lemma *add-sort-inj*[simp]: (*add-sort* *s* = *add-sort* *t*) = (*s* = *t*)
 using *inj-add-sort*[*unfolded inj-def*] by *auto*

lemma *add-sort*: **assumes** *t* : *s* in $\mathcal{T}(C, \mathcal{V})$

shows *sort-of* (*add-sort* *t*) = *s*

using *assms*

proof (*induct*)

case (*Var* *v* *s*)

then show ?*case* by (*cases* *v*, *auto*)

next

case (*Fun* *f* *ts* *ss* *s*)

have *map*: *map* ($\lambda x. \text{sort-of } (\text{add-sort } x)$) *ts* = *ss*

by (*intro* *nth-equalityI*, *insert* *Fun*(3), *auto* *simp*: *list-all2-conv-all-nth*)

show ?*case* **unfolding** *add-sort.simps* *Let-def* *map-map* *o-def* *map*

using *fun-hastypeD*[*OF* *Fun*(1)] by *auto*

qed

definition *rel-term* :: (*'f*, *nat* \times *'s*)*term* \Rightarrow (*'f* \times *'s*, *nat* \times *'s*) *term* \Rightarrow *bool* **where**
rel-term *t* *st* = (*st* = *add-sort* *t*)

definition *rel-smp* :: (*'f*, *'s*)*simple-match-problem-list*

\Rightarrow (*'f*, *'s*)*simple-match-problem-slist*

\Rightarrow (*'f*, *'s*)*simple-match-problem-ms* \Rightarrow *bool* **where**

rel-smp *mpl* *mpsl* *mpm* = (*finite-constructor-form-mp* (*set2* *mpl*))

\wedge *mpsl* = *map* (*map* *add-sort*) *mpl* \wedge *mpm* = *mset* (*map* *mset* *mpl*)

abbreviation *mset2'* **where** *mset2'* *mpl* \equiv *mset* (*map* *mset* *mpl*)

abbreviation *mset3'* **where** *mset3'* *ppl* \equiv *mset* (*map* *mset2'* *ppl*)

lemma *rel-smpD*: **assumes** *rel-smp* *mpl* *mpsl* *mpm*

shows *finite-constructor-form-mp* (*set2* *mpl*)

finite-constructor-form-mp (*mset2* *mpm*)

mpsl = *map* (*map* *add-sort*) *mpl*

set2 *mpl* = *mset2* *mpm*

mpm = *mset2'* *mpl*

using *assms* **unfolding** *rel-smp-def* by (*auto* *simp*: *image-comp*)

definition *large-sort-impl-main* :: (*'f*, *'s*)*simple-pat-problem-slist* \Rightarrow *'s* \Rightarrow (*'f*, *'s*)*simple-pat-problem-slist*

option

where *large-sort-impl-main* $p\ s =$ (*let*
 find-conflict = ($\lambda\ mp.$ ($\exists\ eqc \in\ set\ mp.$ *sort-of* (*hd* *eqc*) = *s*))
 in case partition find-conflict p of
 (*del, keep*) \Rightarrow *if* *del* $\neq\ [] \wedge\ length\ del < cd-sort\ s \wedge$
 ($\forall\ mp \in\ set\ keep.$ $\forall\ eq \in\ set\ mp.$ $\forall\ t \in\ set\ eq.$ $\forall\ x \in\ set\ (vars-term-list$
t). *snd* *x* $\neq\ s$)
 then Some keep
 else None)

definition *large-sort-impl* :: (*f, 's*)*simple-pat-problem-slist* \Rightarrow (*f, 's*)*simple-pat-problem-slist*
option where

large-sort-impl p = (*let*
 terms = concat (concat p);
 sorts = remdups (map sort-of terms)
 in map-option ($\lambda\ (-, -, p', -).$ p') (*extract-option (large-sort-impl-main p) sorts*))

function *simplify-mp-main* :: (*f, 's*)*simple-match-problem-slist* \Rightarrow (*f, 's*)*simple-match-problem-slist*
 \Rightarrow (*f, 's*)*simple-match-problem-slist option where*

simplify-mp-main [] mpout = Some mpout
| *simplify-mp-main (eqc # mp) mpout =* (*if is-singleton-list eqc* \vee *cd-sort (sort-of*
(hd eqc)) = 1
 then simplify-mp-main mp mpout
 else let eqc' = remdups eqc; roots = map aroot eqc'
 in (*if None* \in *set roots*
 then (*if Ball (set eqc')* ($\lambda\ t.$ *Ball (set eqc')* ($\lambda\ s.$ \neg *Conflict-Clash*
(remove-sort s) (remove-sort t))))
 then simplify-mp-main mp (eqc' # mpout) else None)
 else (*if is-singleton-list roots then*
 (*let n = snd (the (hd roots));*
 new-eqcs = zip-lists n (map args eqc')
 in (*if Ball (set new-eqcs)* ($\lambda\ eqc.$ *uniq-list (map sort-of eqc)*) *then*
simplify-mp-main (new-eqcs @ mp) mpout else None)
 else None)))
 by pat-completeness auto

termination

proof (*relation measures* [$\lambda\ mp.$ *sum-list (map* ($\lambda\ eqc.$ *sum-list (map size eqc)*)
(fst mp)), *length o fst*], *force, force, force, goal-cases*)

case (*1 eqc mp mpout eqc' roots n new-eqcs*)

note *new = 1(7)*

from 1 obtain r where *set roots = {r}* **unfolding** *is-singleton-list2* **by** *auto*

with 1(4,6) obtain f where *roots: set roots = {Some (f,n)}*

by (*cases roots; cases r, auto*)

{
 fix t

assume $t \in \text{set } eqc'$
with $1(3-4)$ **obtain** $g\ s\ ts$ **where** $t: t = \text{Fun } (g,s)\ ts$ **and** $rt: \text{Some } (g, \text{length } ts) \in \text{set roots}$
by (*cases t; force simp: aroot-def o-def*)
hence $\text{len}: \text{length } (\text{args } t) = n$ $\text{root } t = \text{Some } ((f,s),n)$ **using** *roots by auto*
hence $\text{size } t = \text{Suc } (\text{sum-list } (\text{map } \text{size } (\text{args } t)) + n)$ **unfolding** t
by (*simp add: size-list-conv-sum-list*)
also have $\text{sum-list } (\text{map } \text{size } (\text{args } t)) = (\sum i \leftarrow [0..<n]. \text{size } (\text{args } t ! i))$
by (*rule arg-cong[of - - sum-list], rule nth-equalityI, insert len, auto*)
finally have $\text{size } t = \text{Suc } n + (\sum i \leftarrow [0..<n]. \text{size } (\text{args } t ! i))$ **by** *simp*
note $\text{len}(1)$ *this*
} **note** $\text{root-terms} = \text{this}$

let $?sum = \lambda eqc. \text{sum-list } (\text{map } \text{size } eqc)$
define common **where** $\text{common} = (\sum t \leftarrow eqc'. \sum i \leftarrow [0..<n]. \text{size } (\text{args } t ! i))$
have $\text{sum-list } (\text{map } ?sum \text{ new-eqcs}) =$
 $\text{sum-list } (\text{map } ?sum (\text{map } (\lambda i. \text{map } (\lambda xs. xs ! i) (\text{map } \text{args } eqc')) [0..<n]))$
unfolding new **by** (*subst zip-lists, insert root-terms(1), auto*)
also have $\dots = (\sum i \leftarrow [0..<n]. (\sum t \leftarrow eqc'. \text{size } (\text{args } t ! i)))$
by (*simp add: size-list-conv-sum-list o-def sum-list-addr sum-list-triv*)
also have $(\sum i \leftarrow [0..<n]. (\sum t \leftarrow eqc'. \text{size } (\text{args } t ! i))) = \text{common}$
unfolding common-def
by (*induct eqc', auto simp: sum-list-addr*)
finally have $\text{lhs}: \text{sum-list } (\text{map } ?sum \text{ new-eqcs}) = \text{common} .$

define sn **where** $sn = \text{Suc } n$
have $?sum eqc' = (\sum t \leftarrow eqc'. \text{Suc } n + (\sum i \leftarrow [0..<n]. \text{size } (\text{args } t ! i)))$
by (*subst map-cong[OF refl], subst root-terms(2), auto*)
also have $\dots = \text{common} + \text{length } eqc' * \text{Suc } n$
unfolding $sn\text{-def}$ [*symmetric*] common-def **by** (*simp add: sum-list-addr sum-list-triv*)
finally have $\text{rhs}: ?sum eqc' = \text{common} + \text{length } eqc' * \text{Suc } n$ **by** *simp*

from 1 **have** $eqc' \neq []$ **unfolding** $is\text{-singleton-list2}$ **by** *auto*
hence $\text{sum-list } (\text{map } ?sum \text{ new-eqcs}) < ?sum eqc'$
unfolding lhs rhs **by** *simp*
also have $\dots \leq ?sum eqc'$ **unfolding** $\langle eqc' = \text{remdups } eqc \rangle$
by (*metis sum.set-conv-list sum-le-sum-list-nat*)
finally show $?case$ **by** *simp*

qed

definition $\text{simplify-mp } mp = \text{simplify-mp-main } mp []$

primrec $\text{simplify-pp} :: ('f, 's)\text{simple-pat-problem-slist} \Rightarrow ('f, 's)\text{simple-pat-problem-slist}$
option **where**
 $\text{simplify-pp } [] = \text{Some } []$
 $| \text{simplify-pp } (mp \# p) = (\text{case } \text{simplify-mp } mp \text{ of } \text{None} \Rightarrow \text{simplify-pp } p$

| *Some mp' ⇒ if mp' = [] then None else map-option (Cons mp') (simplify-pp p)*)

fun *simplify-tpp* :: ('f,'s)tagged-simple-pat-problem-slist ⇒ ('f,'s)tagged-simple-pat-problem-slist option **where**
simplify-tpp (True, p) = Some (True, p)
|i *simplify-tpp (False, p) = map-option (Pair True) (simplify-pp p)*

definition *sτc* :: nat ⇒ nat × 's ⇒ 'f × 's list ⇒ ('f × 's, nat × 's)subst **where**
sτc n x = (λ(f,ss). subst x (Fun (f,snd x) (map Var (indexed-from n ss))))

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

lemma *add-sort-τc*: **assumes** *f : ss → snd x in C*

t : ι in T(C, V)

shows *add-sort (t · τc n x (f,ss)) = add-sort t · sτc n x (f,ss)*

proof –

from *assms(2)*

have *add-sort (t · τc n x (f,ss)) = add-sort t · sτc n x (f,ss) ∧ sort-of (add-sort t · sτc n x (f,ss)) = sort-of (add-sort t)*

proof (*induct*)

case (*Fun g ts*)

thus ?*case apply* (*auto simp: o-def*)

apply (*smt (verit, best) length-map list-all2-conv-all-nth list-eq-iff-nth-eq nth-map*)

by (*smt (verit, del-insts) in-set-conv-nth list-all2-conv-all-nth*)

next

case (*Var y s*)

with *assms(1)* **show** ?*case*

by (*auto simp: sτc-def τc-def subst-def o-def dest: fun-hastypeD split: prod.splits*)

qed

thus ?*thesis by auto*

qed

definition *inst-list* :: nat ⇒ nat × 's ⇒ ('f,'s)simple-pat-problem-slist ⇒ ('f,'s)simple-pat-problem-slist list

where *inst-list n x p = map (λ τ. map (map (map (λt. t · τ))) p) (sτs-list n x)*

definition *full-step* :: nat ⇒ ('f,'s)tagged-simple-pat-problem-slist ⇒ ('f,'s)tagged-simple-pat-problem-slist list + (nat × 's) list list list **where**

full-step n p = (case simplify-tpp p of None ⇒ Inl []

| *Some (True,p') ⇒ (case large-sort-impl p' of*

Some p2 ⇒ (Inl [(True,p2)])

| *None ⇒ (case search-fun-pp p' of*

None ⇒ Inr (map (map (map the-Var)) p')

| *Some (p1,mp,(mp1,eqc,(eqc1,t,eqc2),mp2),p2) ⇒*

let x = the (find is-Var eqc)

in if mp = [[t,x]] ∨ mp = [[x,t]] then Inl (map (Pair False) (inst-list n (the-Var

$x) p')$
 $\text{else let eqn} = \text{eqc1} \ @ \ \text{eqc2}; \text{mpn} = (\text{if length-gt-1 eqn then eqn} \ # \ \text{mp1} \ @ \ \text{mp2} \ \text{else mp1} \ @ \ \text{mp2})$
 $\text{in Inl} \ ((\text{True}, \text{mpn} \ # \ \text{p1} \ @ \ \text{p2})$
 $\# \ \text{map} \ (\text{Pair} \ \text{False}) \ (\text{inst-list} \ n \ (\text{the-Var} \ x) \ ([[x,t]] \ # \ \text{p1} \ @ \ \text{p2}))))))$

definition $\text{fidl-encoder} :: ('x \times 's) \text{list list list} \Rightarrow ('x, 's) \text{fidl-input}$ **where**
 $\text{fidl-encoder } p = (\text{let vars} = \text{remdups} \ (\text{concat} \ (\text{concat} \ p))$
 $\text{in} \ (\text{map} \ (\lambda x. (x, \text{int} \ (\text{cd-sort} \ (\text{snd} \ x)))) \ \text{vars}, \ \text{map} \ (\text{List.maps} \ (\lambda \text{eqc}. \ \text{zip} \ \text{eqc} \ (\text{tl} \ \text{eqc}))) \ p))$

context

fixes $\text{fidl-solver} :: (\text{nat}, 's) \text{fidl-input} \Rightarrow \text{bool}$
begin

definition $\text{fvf-solver} \ \text{fvf} = (\neg \text{fidl-solver} \ (\text{bounds-list} \ (\text{cd-sort} \ \circ \ \text{snd}) \ \text{fvf}, \ \text{dist-pairs-list} \ \text{fvf}))$

partial-function (tailrec) fcf-solver-loop **where**
 $\text{fcf-solver-loop } n \ P = (\text{case } P \ \text{of} \ [] \Rightarrow \text{True}$
 $\mid p \ # \ ps \Rightarrow (\text{case full-step } n \ p \ \text{of}$
 $\text{Inl } ps1 \Rightarrow \text{fcf-solver-loop} \ (n + m) \ (ps1 \ @ \ ps)$
 $\mid \text{Inr } \text{fvf} \Rightarrow \text{if } \text{fvf-solver} \ \text{fvf} \ \text{then } \text{fcf-solver-loop} \ n \ ps$
 $\text{else False}))$

abbreviation add1 **where** $\text{add1} \equiv \text{map} \ \text{add-sort}$

abbreviation add2 **where** $\text{add2} \equiv \text{map} \ \text{add1}$

abbreviation add3 **where** $\text{add3} \equiv \text{map} \ \text{add2}$

definition $\text{fcf-solver-alg} \ n \ p = \text{fcf-solver-loop} \ n \ [(False, \ \text{add3} \ p)]$
end

lemma $\text{mset2-mset2'-set2[simp]}$: $\text{mset2} \ (\text{mset2}' \ mp) = \text{set2} \ mp$
by ($\text{induct} \ mp, \ \text{auto}$)

lemma $\text{mset3-mset3'-set3[simp]}$: $\text{mset3} \ (\text{mset3}' \ p) = \text{set3} \ p$
apply ($\text{induct} \ p$)
apply force
subgoal for $mp \ p$ **using** $\text{mset2-mset2'-set2[of } mp]$ **by** simp
done

end

context $\text{pattern-completeness-context-with-assms}$

begin

lemma mp-steps-cong : **assumes** $\text{finite-constructor-form-pat} \ (\text{mset3} \ (\text{add-mset} \ mp \ p))$
shows $(\rightarrow_{ss})^{**} \ mp \ mp' \ \Longrightarrow$

$(\text{add-mset } (\text{add-mset } mp \ p) \ P, \text{add-mset } (\text{add-mset } mp' \ p) \ P) \in (\Rightarrow_{ss})^*$
 $\wedge \text{finite-constructor-form-pat } (\text{mset3 } (\text{add-mset } mp' \ p))$
proof (*induct rule: rtranclp-induct*)
case (*step mp1 mp2*)
from *spp-simp*[*OF step(2), of p*]
have *p-step*: $\text{add-mset } mp1 \ p \Rightarrow_{ss} \{\#\text{add-mset } mp2 \ p\# \}$.
from *spp-step-ms*[*OF p-step*] *step(3)*
have *fin*: *finite-constructor-form-pat* (*mset3* (*add-mset mp2 p*)) **by** *auto*
from *spp-non-det-step*[*OF p-step*] *step(3)*
have *P-step*: $(\text{add-mset } (\text{add-mset } mp1 \ p) \ P, \{\#\text{add-mset } mp2 \ p\# \} + P) \in \Rightarrow_{ss}$
by *auto*
with *fin step(3)* **show** ?*case by auto*
qed (*insert assms, auto*)

definition *simplified-mp* $mp = (\text{Ball } (\text{set } mp) (\lambda \text{eqc. length eqc} > 1 \wedge (\exists t \in \text{set eqc. is-Var } t) \wedge \text{distinct eqc}))$

definition *simplified-pp* $p = (\text{Ball } (\text{set } p) (\lambda mp. \text{simplified-mp } mp \wedge mp \neq []))$

lemma *simplify-mp-main*: **assumes** *rel-smp mpl (mpsl @ mpsout) mpm*
and *res = simplify-mp-main mpsl mpsout*
and *Ball (set mpsout) prop*
and *tvars-smp (set2 mpl) $\subseteq V$*
and *prop = $(\lambda \text{eqc. length eqc} > 1 \wedge (\exists t \in \text{set eqc. is-Var } t) \wedge \text{distinct eqc})$*
shows *res = Some mpsl' $\implies \exists \text{mpl}' \text{mpm}'. \text{rel-smp } \text{mpl}' \text{mpsl}' \text{mpm}' \wedge (\rightarrow_{ss})^{**}$*
mpm mpm'

$\wedge \text{Ball } (\text{set } \text{mpsl}') \text{ prop} \wedge \text{tvars-smp } (\text{set2 } \text{mpl}') \subseteq V$

$\text{res} = \text{None} \implies \exists \text{mpm}'. (\rightarrow_{ss})^{**} \text{mpm } \text{mpm}' \wedge \text{smp-fail-ms } \text{mpm}'$

using *assms(1-4)*

proof (*atomize(full), induction mpsl mpsout arbitrary: mpl mpm rule: simplify-mp-main.induct*)

case (*1 mpsout*)

then show ?*case unfolding rel-smp-def by auto*

next

case (*2 seqc mps mpsout mpl mpm*)

note *IH = 2.IH*

note *res = 2.prem(2)[unfolded simplify-mp-main.simps(2)]*

note *rel = rel-smpD[OF 2.prem(1)]*

note *props = 2.prem(3)*

from *rel obtain eqc mpl' where mpl: mpl = eqc # mpl' and seqc': seqc = map add-sort eqc*

and *mps: mps @ mpsout = map (map add-sort) mpl' by (cases mpl, auto)*

from *rel have fin: finite-constructor-form-mp (set2 mpl) by auto*

from *fin[unfolded mpl finite-constructor-form-mp-def] obtain t eq where*

eqc: eqc = t # eq by (cases eqc, auto)

with *seqc' have seqc: seqc = add-sort t # map add-sort eq by auto*

from *fin[unfolded mpl eqc finite-constructor-form-mp-def] obtain s where*

t : s in $\mathcal{T}(C, \mathcal{V} \mid SS)$ by auto

hence *ts: t : s in $\mathcal{T}(C, \mathcal{V})$ by (rule typed-restrict-imp-typed)*

from *add-sort[OF ts]*

```

have sort-of (add-sort t) = s by simp
hence sort: sort-of (hd seqc) = s unfolding seqc by simp
note res = res[unfolded sort]
have tvars: tvars-smp (set2 mpl')  $\subseteq V$  tvars-smp ({set eqc})  $\subseteq V$  using 2(7)
unfolding mpl by auto
let ?delcond = is-singleton-list seqc  $\vee$  cd-sort s = 1
show ?case
proof (cases ?delcond)
  case True
    with res have res: res = simplify-mp-main mps mpsout by simp
    from True have is-singleton-list seqc  $\vee$  cd-sort (sort-of (hd seqc)) = 1
      using sort by auto
    note IH = IH(1)[OF this]
    from True have steps:  $(\rightarrow_{ss})^{**}$  mpm (mset (map mset mpl'))
    proof
      assume cd: cd-sort s = 1
      from rel(5)[unfolded mpl eqc]
      have mpm: mpm = add-mset (add-mset t (mset eq)) (mset2' mpl') by auto
      show ?thesis unfolding mpm
        apply (rule r-into-rtranclp)
        apply (rule smp-triv-sort[OF ts cd])
        done
    next
      assume is-singleton-list seqc
      from this[unfolded is-singleton-list2] seqc eqc obtain st where set (map
add-sort eqc) = {st} by auto
      from arg-cong[OF this, of (·) remove-sort, unfolded set-map image-comp
o-def]
      have single: is-singleton-list eqc by (simp add: is-singleton-list2)
      from rel[unfolded mpl] have mpm: mpm = add-mset (mset eqc) (mset2' mpl')
by auto
      define mp where mp = mset2' mpl'
      from single show ?thesis unfolding mpm mp-def[symmetric]
      proof (induct eqc rule: is-singleton-list.induct)
        case (1 x)
          show ?case by (rule r-into-rtranclp, unfold mset.simps, rule smp-singleton)
        next
          case (2 x y xs)
            from 2(2) have y: y = x and single: is-singleton-list (x # xs) by auto
            show ?case unfolding y using 2(1)[OF single] smp-dup[of x]
              by (metis converse-rtranclp-into-rtranclp mset.simps(2))
          qed auto
      qed
      have rel: rel-smp mpl' (mps @ mpsout) (mset2' mpl')
      using rel unfolding rel-smp-def mpl seqc' by (simp add: finite-constructor-form-mp-def)

      from IH[OF rel res - tvars(1)] props steps show ?thesis by (meson rtran-
clp-trans)
    next

```

```

case False
hence ?delcond = False by auto
note res = res[unfolded this if-False]
from False sort have  $\neg (is-singleton-list\ seqc \vee cd-sort\ (sort-of\ (hd\ seqc))) =$ 
1) by auto
note IH = IH(2-)[OF this]
define seqc1 where seqc1 = remdups seqc
note IH = IH[OF seqc1-def]
define xs :: ('f, nat × 's) term list where xs = []
from 2.premis(1)[unfolded mpl seqc'] mps
have rel-smp ((xs @ eqc) # mpl') ((add1 xs @ add1 eqc) # add2 mpl') mpm
by (auto simp: xs-def)
hence  $\exists\ mpm'.\ rel-smp\ ((xs\ @\ remdups\ eqc)\ \#\ mpl')\ ((add1\ xs\ @\ remdups$ 
(add1 eqc)) # add2 mpl') mpm'  $\wedge$  ( $\rightarrow_{ss}$ )** mpm mpm'
proof (induct eqc arbitrary: xs mpm)
case Nil
thus ?case by auto
next
case (Cons x eqc xs mpm)
hence rel: rel-smp ((xs @ x # eqc) # mpl') ((add1 xs @ add1 (x # eqc)) #
add2 mpl') mpm by auto
show ?case
proof (cases x  $\in$  set eqc)
case True
hence rem: remdups (x # eqc) = remdups eqc remdups (add1 (x # eqc)) =
remdups (add1 eqc) by auto
from True have eqc  $\neq$  [] by auto
with rel have rel': rel-smp ((xs @ eqc) # mpl') ((add1 xs @ add1 eqc) #
add2 mpl') (add-mset (mset (xs @ eqc)) (mpm - mset2' [xs @ x # eqc]))
by (auto simp: rel-smp-def finite-constructor-form-mp-def)
from True obtain eqc' where eqc: mset eqc = add-mset x eqc'
by (metis insert-DiffM set-mset-mset)
from rel-smpD(5)[OF rel] eqc have mpm: mpm = add-mset (add-mset x
(add-mset x (mset xs + eqc')) (mset2' mpl')
by auto
have mpm  $\rightarrow_{ss}$  add-mset (add-mset x (mset xs + eqc')) (mset2' mpl')
using smp-dup[of x mset xs + eqc' mset2' mpl', folded mpm] .
hence step: mpm  $\rightarrow_{ss}$  add-mset (mset (xs @ eqc)) (mpm - mset (map mset
[xs @ x # eqc]))
using eqc mpm by auto
show ?thesis unfolding rem using Cons(1)[OF rel'] unfolding rem using
step by auto
next
case False
with inj-add-sort[unfolded inj-def]
have rem: remdups (x # eqc) = x # remdups eqc remdups (add1 (x # eqc))
= add-sort x # remdups (add1 eqc)
by auto
from rel have rel: rel-smp (((xs @ [x]) @ eqc) # mpl') ((add1 (xs @ [x]) @

```

```

add1 eqc) # add2 mpl') mpm
  unfolding rel-def by auto
  show ?thesis unfolding rem using Cons(1)[OF rel] by auto
qed
qed
from this[unfolded xs-def, folded seqc', folded seqc1-def]
obtain mpm1 eqc1 where
  rel: rel-smp (eqc1 # mpl') (seqc1 # map (map add-sort) mpl') mpm1 and
  steps1: ( $\rightarrow_{ss}$ )** mpm mpm1
  by auto
define mpl1 where mpl1 = take (length mps) mpl'
define mpl2 where mpl2 = drop (length mps) mpl'
have mpl': mpl' = mpl1 @ mpl2 unfolding mpl1-def mpl2-def by auto
from mps have mps: mps = add2 mpl1 and mpsout: mpsout = add2 mpl2
  unfolding mpl1-def mpl2-def
  by (force simp add: append-eq-conv-conj take-map) (metis append-eq-conv-conj
drop-map mps)
from rel[unfolded mpl']
have rel-out: rel-smp (mpl1 @ eqc1 # mpl2) (mps @ seqc1 # mpsout) mpm1
unfolding mps mpsout
  by (auto simp: rel-smp-def)

define roots where roots = map aroot seqc1
note IH = IH[OF roots-def]
note res = res[folded seqc1-def, unfolded Let-def, folded roots-def]
show ?thesis
proof (cases None  $\in$  set roots)
  case True
  hence (None  $\in$  set roots) = True by auto
  note res = res[unfolded this if-True]
  note IH = IH(1)[OF True]
  have prop1: prop seqc1 unfolding assms(5)
  proof (intro conjI)
    from True obtain s where s: s  $\in$  set seqc and root: aroot s = None
      unfolding roots-def seqc1-def by auto
    from root have is-Var s unfolding aroot-def by (cases s, auto)
    with s show Bex (set seqc1) is-Var by (auto simp: seqc1-def)
    from False have  $\neg$  is-singleton-list seqc by auto
    from this[unfolded is-singleton-list2]
    have no-single: set seqc  $\neq$  {s} by auto
    with s obtain t where st: {s,t}  $\subseteq$  set seqc1 and s  $\neq$  t
      unfolding seqc1-def by auto
    thus 1 < length seqc1 by (cases seqc1; cases tl seqc1; auto)
    show distinct seqc1 unfolding seqc1-def by auto
  qed
  show ?thesis
proof (cases  $\forall t \in$  set seqc1.  $\forall s \in$  set seqc1. conflicts (remove-sort s) (remove-sort
t)  $\neq$  None)
  case False

```

```

with res have res: res = None by argo
from False obtain ss st where ss: ss ∈ set seqc1 and st: st ∈ set seqc1
  and clash: Conflict-Clash (remove-sort ss) (remove-sort st) by auto
from rel-smpD[OF rel] obtain mpm2 where
  seqc1: seqc1 = map add-sort eqc1 and mpm1: mpm1 = add-mset (mset
eqc1) mpm2
  by auto
from ss st obtain s t where s: s ∈ set eqc1 and t: t ∈ set eqc1 and ss:
ss = add-sort s and
  st: st = add-sort t
  unfolding seqc1 by auto
from clash[unfolded ss st] have clash: Conflict-Clash s t by auto
hence st: s ≠ t by auto
with s t have s ∈# mset eqc1 t ∈# mset eqc1 by auto
from smp-clash[OF clash this] mpm1 have smp-fail-ms mpm1 by auto
with res steps1 show ?thesis by auto
next
case True
with res have res: res = simplify-mp-main mps (seqc1 # mpsout) by auto

{
  fix t
  assume t ∈ set eqc1
  with rel[unfolded rel-smp-def] have add-sort t ∈ set seqc1 by auto
  hence add-sort t ∈ set seqc unfolding seqc1-def by auto
  hence t ∈ set eqc unfolding seqc' by auto
}
hence tvars: tvars-smp (set2 (mpl1 @ eqc1 # mpl2)) ⊆ V using 2(7)
unfolding mpl mpl' by auto
from props prop1 have ∀ eq ∈ set (seqc1 # mpsout). prop eq by auto
from IH[OF True rel-out res this tvars] steps1 props prop1 show ?thesis
  by (meson rtranclp-trans)
qed
next
case roots: False
hence (None ∈ set roots) = False by auto
note res = res[unfolded this if-False]
note IH = IH(2)[OF roots]
from rel-smpD[OF rel]
have fin: finite-constructor-form-mp (set2 (eqc1 # mpl'))
  and eqc1: set eqc1 ∈ mset2 mpm1
  and seqc1: seqc1 = add1 eqc1 by auto
show ?thesis
proof (cases is-singleton-list roots)
case False
with res have res: res = None by auto
from False[unfolded is-singleton-list2] have no-sing: ∄ x. set roots = {x} by
auto
from fin[unfolded finite-constructor-form-mp-def]

```

```

have ne: eqc1 ≠ [] by auto
from ne seqc1 have set roots ≠ {} unfolding roots-def by auto
then obtain f where f: f ∈ set roots by fastforce
with no-sing obtain g where g ∈ set roots - {f} by blast
with f have {f,g} ⊆ set roots and f ≠ g by auto
with roots obtain f g where fg: {Some f, Some g} ⊆ set roots and diff: f
≠ g
  by (cases f; cases g; auto)
  from fg[unfolded roots-def seqc1 map-map o-def aroot]
  obtain s t where st: s ∈ set eqc1 t ∈ set eqc1 and rt: root s = Some f root
t = Some g by auto
  have Conflict-Clash s t using rt diff by (cases s; cases t, auto simp:
conflicts.simps)
  from smp-clash[OF this st eqc1]
  have smp-fail-ms mpm1 .
  with steps1 res show ?thesis by auto
next
case True
from True[unfolded is-singleton-list2] obtain f where set roots = {f} by
auto
  with roots obtain f n where rt: set roots = {Some (f,n)} by (cases f,
auto)
  have set roots = root ' set eqc1 unfolding roots-def seqc1 by auto
  from this[unfolded rt]
  have rt': t ∈# mset eqc1 ⇒ root t = Some (f, n) for t by auto
  have snd: snd (the (hd roots)) = n using rt by (cases roots, auto)
  define new-eqs where new-eqs = zip-lists n (map args seqc1)
  note IH = IH[OF True snd[symmetric] new-eqs-def]
  define new-plain where new-plain = map (λi. map (λx. args x ! i) eqc1)
[0..<n]
  have new-eqs = map (λi. map (λxs. xs ! i) (map args seqc1)) [0..<n]
  unfolding new-eqs-def
  apply (rule zip-lists)
  apply (clarsimp simp add: seqc1)
  subgoal for t using rt'[of t] by (cases t, auto)
  done
  also have ... = map (λi. map (λx. args (add-sort x) ! i) eqc1) [0..<n]
  unfolding seqc1 by (simp add: o-def)
  also have ... = add2 (map (λi. map (λx. args x ! i) eqc1) [0..<n]) unfolding
map-map o-def
  apply (rule map-cong[OF refl], rule map-cong[OF refl])
  subgoal for i t
  using rt'[of t] by (cases t, auto)
  done
  finally have new-eqs: new-eqs = add2 new-plain unfolding new-plain-def .

from rel-smpD(5)[OF rel]
have mpm1: mpm1 = add-mset (mset eqc1) (mset2' mpl') by auto

```

```

from True have (is-singleton-list roots) = True by auto
note res = res[unfolded this if-True, unfolded snd]
note decomp = smp-decomp smp-decomp-fail
note decomp = decomp[OF rt', of mset eqc1, simplified]
have eq: mset2' new-plain = {#{#args t ! i. t ∈# mset eqc1#}. i ∈#
mset-set {0..<n}#}
unfolding mset-map mset-upt image-mset.compositionality o-def new-plain-def
by blast
let ?uniq = ∀ eqc∈set new-eqs. uniq-list (map sort-of eqc)
have ?uniq ↔ (∀ eqc∈set new-plain. UNIQ {sort-of (add-sort x) |. x ∈ set
eqc})
unfolding uniq-list new-eqs set-map by (auto simp: image-comp)
also have ... ↔ (∀ eqc ∈ set new-plain. UNIQ (T(C, V) ' set eqc))
proof (intro ball-cong refl arg-cong[of - - UNIQ])
fix eqc
assume eqc ∈ set new-plain
from this[unfolded new-plain-def, simplified] obtain i where i: i < n
and eqc: eqc = map (λxa. args xa ! i) eqc1 by auto
have UNIQ {sort-of (add-sort x) |. x ∈ set eqc} = UNIQ {Some (sort-of
(add-sort x)) |. x ∈ set eqc}
unfolding Uniq-def by auto
also have {Some (sort-of (add-sort x)) |. x ∈ set eqc} = T(C, V) ' set eqc
proof (intro image-cong refl)
fix ti
assume ti ∈ set eqc
from this[unfolded eqc] obtain t where t: t ∈ set eqc1 and ti: ti = args
t ! i by auto
from fin t obtain s where t : s in T(C, V |' SS) unfolding fi-
nite-constructor-form-mp-def by auto
hence typed: t : s in T(C, V) by (rule typed-restrict-imp-typed)
from rt'[of t] t i obtain ts where tf: t = Fun f ts and tis: ti ∈ set ts
unfolding ti by (cases t, auto)
from typed[unfolded tf] tis obtain si where typed: ti : si in T(C, V)
by (meson Fun-in-dom-imp-arg-in-dom in-dom-iff-ex-type)
from add-sort[OF this] have sort-of (add-sort ti) = si by auto
with typed ti t i show Some (sort-of (add-sort ti)) = T(C, V) ti
by (force simp: hastype-def eqc)
qed
finally show UNIQ {sort-of (add-sort x) |. x ∈ set eqc} = UNIQ (T(C, V)
' set eqc) .
qed
finally have uniq: (∀ eqc∈set new-eqs. uniq-list (map sort-of eqc))
= (∀ eqc∈set new-plain. UNIQ (T(C, V) ' set eqc)) .
show ?thesis
proof (cases ?uniq)
case True
with res
have res: res = simplify-mp-main (new-eqs @ mps) mpsout
by (auto simp: new-eqs-def)

```

```

      from True have uniq: eq ∈# mset (map mset new-plain) ⇒ UNIQ
(T(C, V) ‘ set-mset eq) for eq
      unfolding uniq by auto
      note IH = IH[OF True - res]
      from decomp(1)[OF eq uniq]
      have step2: mpm1 →ss mset2' new-plain + mset2' mpl' by (auto simp:
mpm1)
      from rel-smpD[OF 2.prem(1)]
      have fin: finite-constructor-form-pat (mset3 {#mpm#})
        by (auto simp: finite-constructor-form-pat-def)
      from steps1 step2
      have steps2: (→ss)** mpm (mset2' (new-plain @ mpl')) by auto
      from mp-steps-cong[OF fin steps2, of {#}, THEN conjunct2]
      have fin': finite-constructor-form-mp (set2 (new-plain @ mpl'))
        by (auto simp: finite-constructor-form-pat-def image-Un image-comp)
      hence rel-new: rel-smp (new-plain @ mpl1 @ mpl2) ((new-egs @ mps) @
mpsout)
        (mset2' (new-plain @ mpl'))
      unfolding mps new-egs seqc1 mpsout rel-smp-def mpl' by auto
      have tvars-smp (set2 (mpl1 @ mpl2)) ⊆ V using mpl' tvars by auto
      moreover have tvars-smp (set2 new-plain) ⊆ V unfolding new-plain-def
      proof (clarsimp, goal-cases)
        case (1 x ι i t)
        hence mem: (x,ι) ∈ vars t using rt'[of t] by (cases t, auto)
        from ⟨t ∈ set eqc1⟩
        have add-sort t ∈ set (add1 eqc1) by auto
        also have set (add1 eqc1) = set (add1 eqc)
          unfolding seqc1[symmetric] seqc1-def set-remdups seqc' by simp
        finally have t ∈ set eqc by auto
        with mem tvars(2) show (x,ι) ∈ V by auto
      qed
      ultimately have tvars: tvars-smp (set2 (new-plain @ mpl1 @ mpl2)) ⊆
V by auto
      from IH[OF rel-new - tvars] props steps2 show ?thesis
        by (metis (no-types, lifting) rtranclp-trans)
    next
      case False
      with res have res: res = None by (auto simp: new-egs-def)
      from False[unfolded uniq, unfolded new-plain-def, simplified]
      obtain i where i: i < n and uniq: ¬ UNIQ (T(C, V) ‘ {args t ! i |. t ∈
set eqc1}) by auto
      from decomp(2)[OF i uniq] have smp-fail-ms mpm1 unfolding mpm1
by auto
      with steps1 res show ?thesis by auto
    qed
  qed
qed
qed
qed

```

```

lemma sorts-of-subterm:  $t \supseteq a \implies a : s \text{ in } \mathcal{T}(C, \mathcal{V}) \implies s \in \text{sorts-of } (\text{add-sort } t)$ 
proof (induct rule: supteq.induct)
  case (refl t)
  then show ?case using add-sort[of t s] by (cases t; force)
next
  case (subt u ss t f)
  then show ?case by auto
qed

lemma vars-term-add-sort[simp]: vars-term (add-sort t) = vars-term t
  by (induct t, auto)

lemma eroot-add-sort: assumes  $t : \iota \text{ in } \mathcal{T}(C, \mathcal{V})$ 
  shows eroot (add-sort t) = map-sum (map-prod ( $\lambda f. (f, \iota)$ ) id) id (eroot t)
  using assms add-sort[OF assms]
  by (cases t, auto)

lemma large-sort-impl: assumes large-sort-impl (add3 p) = Some ap'
  and fin: finite-constructor-form-pat (set3 p)
  and simpl: simplified-pp p
  shows  $\exists p'. \text{mset3}' p \Rightarrow_{ss} \{\# \text{mset3}' p' \# \} \wedge ap' = \text{add3 } p' \wedge \text{set } p' \subseteq \text{set } p \wedge$ 
  length p'  $\leq$  length p
proof –
  from assms(1)[unfolded large-sort-impl-def Let-def]
  obtain bef s aft sorts where
    extract-option (large-sort-impl-main (add3 p)) sorts = Some (bef, s, ap', aft)
    and sorts: sorts = remdups (map sort-of (concat (concat (add3 p))))
  by auto
  from extract-option-Some[OF this(1)]
  have large: large-sort-impl-main (add3 p) s = Some ap' and s:  $s \in \text{set } \text{sorts}$  by
  auto
  {
    from s[unfolded sorts, simplified] obtain mp eqc t where
      *: mp  $\in \text{set } p$  eqc  $\in \text{set } mp$  t  $\in \text{set } eqc$  and st:  $s = \text{sort-of } (\text{add-sort } t)$  by
  blast
  from * fin[unfolded finite-constructor-form-pat-def finite-constructor-form-mp-def]
  obtain  $\iota$  where fin: finite-sort C  $\iota$  and t:  $t : \iota \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS)$  by fastforce
  hence iota:  $\iota \in S$  by (metis typed-imp-S)
  with s st t have  $s = \iota$ 
  by (meson add-sort typed-restrict-imp-typed)
  with fin iota have  $s \in S$  finite-sort C s by auto
  } note  $s = \text{this}$ 

define confl where confl = ( $\lambda mp :: ('f \times 's, \text{nat} \times 's)$  term list list.  $\exists eqc \in \text{set}$ 
  mp.  $\text{sort-of } (\text{hd } eqc) = s$ )
obtain del keep where part1: partition (confl o add2) p = (del, keep) by auto
hence part2: partition confl (add3 p) = (map add2 del, map add2 keep)

```

```

unfolding partition-filter-conv by (simp add: comp-assoc filter-map)
from large[unfolded large-sort-impl-main-def, folded confl-def, unfolded Let-def
part2 split]
have card: length del < cd-sort s and del: del ≠ [] and
  keep: ( $\forall mp \in \text{set } (add3 \text{ keep}). \forall eq \in \text{set } mp. \forall t \in \text{set } eq. \forall x \in \text{set } (\text{vars-term-list } t).$ 
snd x ≠ s)
  and ap': ap' = (add3 keep)
  by (auto split: if-splits)
{
  fix i
  assume i: i < length del
  let ?mp = del ! i
  from i have ?mp ∈ set del by auto
  with part1 have confl: confl (add2 ?mp) and ?mp ∈ set p by auto
  with simpl[unfolded simplified-pp-def] fin[unfolded finite-constructor-form-pat-def]
  have simpl: simplified-mp ?mp and fin: finite-constructor-form-mp (set2 ?mp)
by auto
  from confl[unfolded confl-def]
  obtain eqc where eqc: eqc ∈ set ?mp and sort: sort-of (hd (add1 eqc)) = s
by auto
  from simpl[unfolded simplified-mp-def, rule-format, OF eqc] obtain x
  where len: 1 < length eqc and var: Var x ∈ set eqc and dist: distinct eqc
by auto
  from fin[unfolded finite-constructor-form-mp-def, rule-format, of set eqc] eqc
  obtain ι where same-sort:  $\bigwedge t. t \in \text{set } eqc \implies t : \iota$  in  $\mathcal{T}(C, \mathcal{V} \mid SS)$  by auto
  from len have hd eqc ∈ set eqc and hd: hd (add1 eqc) = add-sort (hd eqc) by
(cases eqc, auto)+
  from same-sort[OF this(1)] sort have ι = s unfolding hd
  by (metis add-sort typed-restrict-imp-typed)
  note same-sort = same-sort[unfolded this]
  from split-list[OF var] obtain bef aft where split: eqc = bef @ Var x # aft
by auto
  with len obtain t where t ∈ set bef ∪ set aft by (cases bef; cases aft, auto)
  with split dist have xt: Var x ≠ t t ∈ set eqc by auto
  from same-sort[OF var] have xs: snd x = s
  by (simp add: hastype-restrict)
  from same-sort[OF xt(2)] have sorted: t : s in  $\mathcal{T}(C, \mathcal{V})$ 
  by (metis typed-restrict-imp-typed)
  have  $\exists x t \text{ eqc. } \{Var x, t\} \subseteq \text{set } eqc \wedge Var x \neq t \wedge eqc \in \text{set } ?mp \wedge \text{snd } x =$ 
s ∧ t : s in  $\mathcal{T}(C, \mathcal{V})$ 
  by (rule exI[of - x], rule exI[of - t], rule exI[of - eqc], insert xt var eqc xs
sorted, auto)
}
hence  $\forall i. \exists x t \text{ eqc. } i < \text{length } del \implies \{Var x, t\} \subseteq \text{set } eqc \wedge Var x \neq t \wedge eqc$ 
 $\in \text{set } (del ! i) \wedge \text{snd } x = s \wedge t : s$  in  $\mathcal{T}(C, \mathcal{V})$ 
by blast
from choice[OF this] obtain x where
   $\forall i. \exists t \text{ eqc. } i < \text{length } del \implies \{Var (x i), t\} \subseteq \text{set } eqc \wedge Var (x i) \neq t \wedge eqc$ 
 $\in \text{set } (del ! i) \wedge \text{snd } (x i) = s \wedge t : s$  in  $\mathcal{T}(C, \mathcal{V})$ 

```

by blast
from choice[*OF this*] **obtain** t **where**
 $\forall i. \exists eqc. i < \text{length } del \longrightarrow \{ \text{Var } (x \ i), t \ i \} \subseteq \text{set } eqc \wedge \text{Var } (x \ i) \neq t \ i \wedge$
 $eqc \in \text{set } (del \ ! \ i) \wedge \text{snd } (x \ i) = s \wedge t \ i : s \text{ in } \mathcal{T}(C, \mathcal{V})$
by blast
from choice[*OF this*] **obtain** eqc **where**
 $xte: \bigwedge i. i < \text{length } del \implies \{ \text{Var } (x \ i), t \ i \} \subseteq \text{set } (eqc \ i) \wedge \text{Var } (x \ i) \neq t \ i$
 $\wedge (eqc \ i) \in \text{set } (del \ ! \ i) \wedge \text{snd } (x \ i) = s \wedge t \ i : s \text{ in } \mathcal{T}(C, \mathcal{V})$
by blast
let $?Var = \text{Var} :: \text{nat} \times 's \Rightarrow ('f, \text{nat} \times 's)\text{term}$
define n **where** $n = \text{length } del - 1$
from del **have** $id: i < \text{length } del \longleftrightarrow i \leq n$ **for** i **unfolding** $n\text{-def}$ **by** (*cases*
 del, auto)
have $\text{card } (t \ ' \ {0..n}) \leq \text{card } \{0..n\}$
by (*intro card-image-le, auto*)
also have $\dots = \text{Suc } n$ **by** *auto*
also have $\text{Suc } n = \text{length } del$ **unfolding** $n\text{-def}$ **using** del **by** (*cases del, auto*)
also have $\dots < \text{cd-sort } s$ **by** *fact*
also have $\dots = \text{min } k (\text{card-of-sort } C \ s)$ **unfolding** $cd[OF \ s(1)]$ **by** *auto*
also have $\dots \leq \text{card-of-sort } C \ s$ **by** *auto*
finally have $\text{card } (t \ ' \ {0..n}) < \text{card-of-sort } C \ s$ **by** *auto*
{
 fix y
 assume $y \in \text{tvars-spat } (mset3 \ (mset3' \ \text{keep}))$
 hence $y \in \text{tvars-spat } (\text{set3 } \text{keep})$ **by** *simp*
 with keep **have** $y: \text{snd } y \neq s$ **by** *auto*
 have $\text{Var } y \notin (t \ ' \ \{0..n\} \cup (\text{Var} \circ x) \ ' \ \{0..n\})$
 proof
 assume $\text{Var } y \in (t \ ' \ \{0..n\} \cup (\text{Var} \circ x) \ ' \ \{0..n\})$
 then obtain i **where** $i: i \leq n$ **and** $ti: t \ i = \text{Var } y \vee x \ i = y$ **by** *auto*
 from $xte[\text{unfolded } id, OF \ i]$ ti **have** $\text{snd } y = s$ **by** *auto*
 with y **show** *False* **by** *auto*
 qed
} **note** $\text{disjoint} = \text{this}$
have $\text{Sucn}: \text{Suc } n = \text{length } del$ **unfolding** $n\text{-def}$ **using** del **by** (*cases del, auto*)
have $\text{step}: mset3' \ \text{keep} + mset (\text{map } (mset2' \ o \ (!) \ del) [0..< \ \text{Suc } n]) \Rightarrow_{ss} \{\#$
 $mset3' \ \text{keep} \ \#\}$
 apply (*rule spp-delete-large-sort[of n x s mset o eqc - t]*)
 subgoal for i **using** $xte[\text{unfolded } id, of \ i]$ **by** *auto*
 subgoal using disjoint **by** *force*
 subgoal by *fact*
 done
from part1 **have** $mset \ p = mset \ del + mset \ \text{keep}$ **by** *auto*
hence $mset3' \ p = mset3' \ \text{keep} + mset3' \ del$ **by** *simp*
also have $del = \text{map } ((!) \ del) [0..< \ \text{length } del]$
 by (*intro nth-equalityI, auto*)
also have $mset3' \ \dots = mset (\text{map } (mset2' \ o \ (!) \ del) [0..< \ \text{Suc } n])$ **unfolding**
 $\text{Sucn } o\text{-def}$
 unfolding $mset\text{-map image-mset.compositionality } o\text{-def} \dots$

finally have $step: mset3' p \Rightarrow_{ss} \{\# mset3' keep \#\}$ **using** *step* **by** *auto*
show *?thesis*
by (*rule exI*, *intro conjI*, *rule step*, *rule ap'*, *insert part1*, *auto*)
qed

lemma *simplify-mp*: **assumes** *finite-constructor-form-mp* (*set2 mp*)
and *res*: $res = simplify-mp (add2 mp)$
and *vars*: $tvars-smp (set2 mp) \subseteq V$
shows $res = Some amp' \implies$
 $\exists mp'. amp' = add2 mp' \wedge (\rightarrow_{ss})^{**} (mset2' mp) (mset2' mp') \wedge simplified-mp$
 $amp' \wedge tvars-smp (set2 mp') \subseteq V$
 $res = None \implies \exists mp'. (\rightarrow_{ss})^{**} (mset2' mp) mp' \wedge smp-fail-ms mp'$
proof –
from *res[unfolded simplify-mp-def]*
have *res*: $res = simplify-mp-main (add2 mp) []$ (**is** $- = simplify-mp-main - ?Nil$)
by *auto*
have *all*: $\forall eqc \in set \ ?Nil. P eqc$ **for** *P* **by** *auto*
have *rel-smp mp* ($add2 mp @ []$) (*mset2' mp*)
unfolding *rel-smp-def* **using** *assms(1)* **by** *blast*
note *main* = *simplify-mp-main[OF this res - vars refl, OF all, folded simplified-mp-def]*
show $res = None \implies \exists mp'. (\rightarrow_{ss})^{**} (mset2' mp) mp' \wedge smp-fail-ms mp'$
using *main(2)* **by** *auto*
assume $res = Some amp'$
from *main(1)[OF this]* **obtain** $mp' mpm'$
where *rel*: *rel-smp mp' amp' mpm'* **and** *steps*: $(\rightarrow_{ss})^{**} (mset2' mp) mpm'$
and *smp*: *simplified-mp amp'*
and *vars*: $tvars-smp (set2 mp') \subseteq V$
by *auto*
show $\exists mp'. amp' = add2 mp' \wedge (\rightarrow_{ss})^{**} (mset2' mp) (mset2' mp') \wedge simplified-mp$
 $amp' \wedge tvars-smp (set2 mp') \subseteq V$
proof (*intro exI[of - mp'] conjI steps smp vars*)
from *rel-smpD[OF rel] steps*
show $amp' = add2 mp' (\rightarrow_{ss})^{**} (mset (map mset mp)) (mset (map mset mp'))$

by *auto*
qed
qed

lemma *simplify-pp*: **assumes** *finite-constructor-form-pat* (*set3 p*)
and $res = simplify-pp (add3 p)$
and $tvars-spat (set3 p) \subseteq V \wedge length p < k$
shows $res = Some ap' \implies \exists p'. ap' = add3 p'$
 $\wedge (add-mset (mset3' p) P, add-mset (mset3' p') P) \in (\Rightarrow_{ss})^*$
 $\wedge simplified-pp ap'$
 $\wedge finite-constructor-form-pat (set3 p')$
 $\wedge tvars-spat (set3 p') \subseteq V \wedge length p' < k$
(is $?A \implies ?B$ **)**
and $res = None \implies (add-mset (mset3' p) P, P) \in (\Rightarrow_{ss})^* (\mathbf{is} \ ?C \implies ?D)$

proof –

```

define p2 :: ('f, nat × 's) term list list list where p2 = []
from assms have fin: finite-constructor-form-pat (set3 (p @ p2)) unfolding
p2-def by auto
from assms have res: res = map-option ((@) (add3 p2)) (simplify-pp (add3 p))
by (cases simplify-pp (add3 p), auto simp: p2-def)
have smp: simplified-pp (add3 p2) unfolding p2-def simplified-pp-def by auto
have vars: tvars-spat (set3 (p @ p2)) ⊆ V ∧ length (p @ p2) < k using assms(3)
unfolding p2-def by auto
have main: res = Some ap' ⇒ ∃ p'. ap' = add3 p' ∧
(add-mset (mset3' (p @ p2)) P, add-mset (mset3' p') P) ∈ (⇒ss)* ∧ simplified-pp ap'
∧ finite-constructor-form-pat (set3 p') ∧ tvars-spat (set3 p') ⊆ V ∧ length
p' < k
res = None ⇒ (add-mset (mset3' (p @ p2)) P, P) ∈ (⇒ss)*
using fin res smp vars
proof (atomize(full), induct p arbitrary: ap' p2 res)
case (Nil res)
thus ?case by (auto simp: simplified-pp-def)
next
case (Cons mp p ap' p2 res)
have res: res = map-option ((@) (add3 p2)) (simplify-pp (add3 (mp # p)))
using Cons by auto
from Cons(2) have fin: finite-constructor-form-mp (set2 mp)
and finp: finite-constructor-form-pat (set3 (p @ p2))
and fin-both: finite-constructor-form-pat (mset3 (add-mset (mset2' mp)
(mset3' p + mset3' p2)))
by (auto simp: finite-constructor-form-pat-def image-comp)
from Cons(5) have tv-mp: tvars-smp (set2 mp) ⊆ V
and tv-pp2: tvars-spat (set3 (p @ p2)) ⊆ V ∧ length (p @ p2) < k
and k: Suc (length (p @ p2)) < k by auto
show ?case
proof (cases simplify-mp (add2 mp))
case None
with res have res: res = map-option ((@) (add3 p2)) (simplify-pp (add3 p))
by auto
from simplify-mp(2)[OF fin refl tv-mp None]
obtain mp' where steps: (→ss)** (mset2' mp) mp' and fail: smp-fail-ms
mp' by auto
from mp-steps-cong[OF fin-both steps, of P]
have steps: (add-mset (mset3' (mp # p) + mset3' p2) P, add-mset (add-mset
mp' (mset3' p + mset3' p2)) P) ∈ (⇒ss)*
and fin': finite-constructor-form-pat (mset3 (add-mset mp' (mset3' p +
mset3' p2)))
by auto
have (add-mset (add-mset mp' (mset3' p + mset3' p2)) P, {#mset3' p +
mset3' p2#} + P) ∈ ⇒ss
by (rule spp-non-det-step, rule spp-delete[OF fail], insert fin', auto)
with steps have steps: (add-mset (mset3' (mp # p @ p2)) P, add-mset

```

```

(mset3' (p @ p2)) P) ∈ (⇒ss)* by auto
  note IH = Cons(1)[OF finp res Cons(4) tv-pp2]
  from IH steps res show ?thesis by (cases simplify-pp (add3 p); force)
next
  case (Some amp')
  from res Some
  have res: res = map-option ((@) (add3 p2))
    (if amp' = [] then None else map-option ((#) amp') (simplify-pp (add3
p))) by auto
  from simplify-mp(1)[OF fin refl tv-mp Some] obtain mp' where
    eq: amp' = add2 mp' and steps: (→ss)** (mset2' mp) (mset2' mp') and
smp: simplified-mp amp'
    and tvmp': tvars-smp (set2 mp') ⊆ V
    by auto
  have fin-cong: finite-constructor-form-pat (mset3 (add-mset (mset2' mp)
(mset3' (p @ p2))))
    using fin fin-both by auto
  from mp-steps-cong[OF fin-both steps, of P]
  have steps: (add-mset (mset3' (mp # (p @ p2))) P, add-mset (mset3' (mp'
# (p @ p2))) P) ∈ (⇒ss)*
    and fin': finite-constructor-form-pat (mset3 (mset3' (mp' # p @ p2)))
    by auto
  show ?thesis
  proof (cases amp' = [])
    case True
    with res have res: res = None by auto
    from True eq have mp': mp' = [] by auto
    have step: mset3' (mp' # (p @ p2)) ⇒ss {#} unfolding mp'
      by (simp, rule spp-solved)
    have (add-mset (mset3' (mp' # (p @ p2))) P, {#} + P) ∈ ⇒ss
      by (rule spp-non-det-step[OF step fin'])
    with steps res
    show ?thesis by auto
  next
  case False
    with res eq have res: res = map-option ((@) (add3 (p2 @ [mp'])))
(simplify-pp (add3 p))
      by (cases simplify-pp (add3 p), auto)
    from Cons(4) smp eq False eq have simplified-pp (add3 (p2 @ [mp'])) by
(auto simp: simplified-pp-def)
    note IH = Cons(1)[OF - res this]
    from fin' have fin'': finite-constructor-form-pat (set3 (p @ p2 @ [mp']))
      by (simp add: image-comp image-Un)
    note IH = IH[OF fin'] tvmp' tv-pp2 k
    with IH steps res show ?thesis by (cases simplify-pp (add3 (p @ p2)));
force)
  qed
  qed
  qed

```

from $\text{main}(1)[\text{unfolded } p2\text{-def}]$ **show** $?A \implies ?B$ **by** *auto*
from $\text{main}(2)[\text{unfolded } p2\text{-def}]$ **show** $?C \implies ?D$ **by** *auto*
qed

lemma *inst-list-result*: **assumes** *finite-constructor-form-pat* ($\text{set3 } p$)
shows $\text{inst-list } n \ x \ (\text{add3 } p)$
 $= \text{map } \text{add3} \ (\text{map } (\lambda \tau. \ (\text{map } (\text{map } (\text{map } (\lambda t. \ t \cdot \tau))) \ p)) \ (\tau\text{-list } n \ x))$
unfolding *map-map o-def s τ s-list-def τ s-list-def inst-list-def*
proof (*intro map-cong refl, goal-cases*)
case ($1 \ fs \ mp \ eqc \ t$)
from $1 \ \text{assms}$ **have** *finite-constructor-form-mp* ($\text{set2 } mp$) **by** (*auto simp: finite-constructor-form-pat-def*)
with $1 \ \text{obtain } s \ \text{where } t : s \ \text{in } \mathcal{T}(C, \mathcal{V} \mid 'SS)$ **by** (*auto simp: finite-constructor-form-mp-def*)
hence $t : t : s \ \text{in } \mathcal{T}(C, \mathcal{V})$ **by** (*rule typed-restrict-imp-typed*)
from $1[\text{unfolded } Cl]$ **obtain** $f \ ss$ **where** $fs: fs = (f, ss)$ **and** $f: f : ss \rightarrow \text{snd } x \ \text{in } C$ **by** *auto*
show $\text{add-sort } t \cdot s\tau c \ n \ x \ fs = \text{add-sort } (t \cdot \tau c \ n \ x \ fs)$ **using** *add-sort- τc [OF f t]*
 fs **by** *auto*
qed

lemma *inst-list*: **assumes** $\{\#\{\# \text{Var } x, t\#\}\# \} \in \# \ \text{mset3}' \ p$
is-Fun t
 $\text{tvvars-spat } (\text{set3 } p) \subseteq \{..\lt n\} \times UNIV \wedge \text{length } p < k$
finite-constructor-form-pat ($\text{set3 } p$)
shows $\exists \ ps'. \ \text{mset3}' \ p \Rightarrow_{ss} \ \text{mset} \ (\text{map } \text{mset3}' \ ps') \wedge \ \text{map } \text{add3} \ ps' = \text{inst-list } n \ x$
 $(\text{add3 } p)$
 $\wedge \ \text{Ball } (\text{set } ps') \ (\lambda p'. \ \text{tvvars-spat } (\text{set3 } p') \subseteq \{..\lt n+m\} \times UNIV \wedge \text{length } p' < k)$
proof –
note *comp = o-def image-comp mset-map set-mset-mset image-mset.compositionality set-image-mset*
have $\text{fst } ' \ \text{tvvars-spat} \ (\text{mset3} \ (\text{mset3}' \ p)) \cap \{n..\lt n + m\} = \{\}$
using *assms(3) unfolding comp by fastforce*
from *spp-inst[OF assms(1–2) this]*
have *step: mset3' p \Rightarrow_{ss} mset (map (lambda tau. image-mset (image-mset (image-mset (lambda t. t . tau))) (mset3' p)) (tau-list n x)) .*
 $\{$
fix $t \ eqc \ mp \ \tau$
assume $t: t \in \text{set } eqc \ eqc \in \text{set } mp \ mp \in \text{set } p$ **and** $\text{tau}: \tau \in \text{set } (\tau\text{-list } n \ x)$
from $t \ \text{assms}(3)$ **have** $\text{fst } ' \ \text{vars } t \subseteq \{..\lt n\}$ **by** *fastforce*
with tau **have** $\text{fst } ' \ \text{vars } (t \cdot \tau) \subseteq \{..\lt n\} \cup \{..\lt n+m\}$
by (*auto simp add: τ s-list τ s-def τc -def subst-def vars-term-subst in-set-indexed-from-eq dest!: m split: if-splits simp: set-zip*)
hence $\text{fst } ' \ \text{vars } (t \cdot \tau) \subseteq \{..\lt n + m\}$ **by** *auto*
 $\}$ **note** $\text{vars} = \text{this}$
show *?thesis*
apply (*intro exI, rule conjI[OF - conjI[OF inst-list-result[OF assms(4), symmetric]]]*)

```

    apply (unfold comp, rule step[unfolded comp])
    apply (intro ballI conjI)
    apply clarsimp subgoal for tau x s mp eqc t using vars[of t eqc mp tau] by
auto
    using assms by auto
qed

lemma simplified-mp-add2: simplified-mp (add2 mp) = simplified-mp mp
  unfolding simplified-mp-def
  by (auto simp: distinct-map inj-on-def)

lemma simplified-pp-add3: simplified-pp (add3 p) = simplified-pp p
  unfolding simplified-pp-def by (simp add: simplified-mp-add2)

fun simpl-tag :: ('f,'s)tagged-simple-pat-problem-slist  $\Rightarrow$  bool where
  simpl-tag (False,p) = True
| simpl-tag (True,p) = simplified-pp p

lemma simplify-tpp: assumes inv: simpl-tag (tag, add3 p)
  and res: res = simplify-tpp (tag, add3 p)
  and fin: finite-constructor-form-pat (set3 p)
  and vars: tvars-spat (set3 p)  $\subseteq$  V  $\wedge$  length p < k
shows res = Some (tag',ap')  $\Longrightarrow$   $\exists$  p'. ap' = add3 p'
   $\wedge$  (add-mset (mset3' p) P, add-mset (mset3' p) P)  $\in$  ( $\Rightarrow_{ss}$ )*
   $\wedge$  tag' = True
   $\wedge$  simpl-tag (tag',ap')
   $\wedge$  finite-constructor-form-pat (set3 p')
   $\wedge$  tvars-spat (set3 p')  $\subseteq$  V  $\wedge$  length p' < k
  and res = None  $\Longrightarrow$  (add-mset (mset3' p) P, P)  $\in$  ( $\Rightarrow_{ss}$ )*
proof (atomize(full), goal-cases)
  case 1
  show ?case
  proof (cases tag)
    case True
    thus ?thesis using assms by auto
  next
  case False
  show ?thesis
  proof (cases simplify-pp (add3 p))
    case None
    from simplify-pp(2)[OF fin refl vars None, of P] None res
    show ?thesis using False by auto
  next
  case (Some ap')
  from simplify-pp(1)[OF fin refl vars Some, of P] Some res
  show ?thesis using False by (auto simp: simplified-pp-add3)
  qed
qed
qed

```

abbreviation add_4 **where** $add_4 \equiv \text{map } (\text{map-prod } id \text{ } add_3)$

abbreviation $mset_4'$ **where** $mset_4' \equiv \text{image-mset } (mset_3' \text{ } o \text{ } snd) \text{ } o \text{ } mset$

lemma *full-step*: **assumes** $tvarsp: tvarsp\text{-spat } (set_3 \text{ } p) \subseteq \{..<n\} \times UNIV \wedge \text{length } p < k$

and $finp: \text{finite-constructor-form-pat } (set_3 \text{ } p)$

and $inv\text{-tag}: \text{simpl-tag } (tag, add_3 \text{ } p)$

and $result: \text{full-step } n \text{ } (tag, add_3 \text{ } p) = res$

shows $res = \text{Inl } aps \implies \exists ps. aps = add_4 \text{ } ps$

$\wedge (add\text{-mset } (mset_3' \text{ } p) \text{ } P, mset_4' \text{ } ps + P) \in (\equiv_{ss})^+$

$\wedge \text{Ball } (snd \text{ } ' \text{ } set \text{ } ps) (\lambda p'. tvarsp\text{-spat } (set_3 \text{ } p') \subseteq \{..<n+m\} \times UNIV$

$\wedge \text{length } p' < k)$

$\wedge \text{Ball } (set \text{ } aps) \text{ } simpl\text{-tag}$

$res = \text{Inr } fvf \implies \text{simplified-pp } (\text{map } (\text{map } (\text{map } (Var :: - \implies ('f,-)term)))) \text{ } fvf) \wedge$

$\text{length } fvf < k$

$\wedge (add\text{-mset } (mset_3' \text{ } p) \text{ } P, add\text{-mset } (mset_3' \text{ } (\text{map } (\text{map } (\text{map } Var))) \text{ } fvf)) \text{ } P)$

$\in (\equiv_{ss})^*$

$\wedge (\forall x \iota. (x, \iota) \in set \text{ } (\text{concat } (\text{concat } fvf))) \longrightarrow \text{card-of-sort } C \iota < k)$

proof (*atomize(full), goal-cases*)

case 1

note $res = \text{result}[\text{symmetric}, \text{unfolded full-step-def}]$

show *?case*

proof (*cases simplify-tp (tag, add3 p)*)

case *None*

with res **have** $res: res = \text{Inl } []$ **by** *auto*

from *simplify-tp(2)[OF inv-tag refl finp tvarsp None, of P]*

have $(add\text{-mset } (mset_3' \text{ } p) \text{ } P, P) \in (\equiv_{ss})^*$

hence $(add\text{-mset } (mset_3' \text{ } p) \text{ } P, P) \in (\equiv_{ss})^+$

by (*simp add: rtrancl-eq-or-trancl*)

with res **show** *?thesis* **by** *auto*

next

case (*Some tap1*)

then obtain $tag' \text{ } ap1$ **where** $Some: \text{simplify-tp } (tag, add_3 \text{ } p) = \text{Some } (tag',$

$ap1)$ **by** (*cases tap1, auto*)

from *simplify-tp(1)[OF inv-tag refl finp tvarsp Some, of P]*

obtain p' **where** $ap1: ap1 = add_3 \text{ } p'$

and $steps': (add\text{-mset } (mset_3' \text{ } p) \text{ } P, add\text{-mset } (mset_3' \text{ } p') \text{ } P) \in (\equiv_{ss})^*$

and $finp': \text{finite-constructor-form-pat } (set_3 \text{ } p')$

and $simpl': \text{simplified-pp } (add_3 \text{ } p')$

and $tag': tag' = \text{True}$

and $tvarsp': tvarsp\text{-spat } (set_3 \text{ } p') \subseteq \{..<n\} \times UNIV \wedge \text{length } p' < k$

by *auto*

note $res = \text{res}[\text{unfolded } \text{Some } \text{option.simps } ap1 \text{ } tag' \text{ } \text{split } \text{bool.simps}]$

show *?thesis*

proof (*cases large-sort-impl (add3 p')*)

case (*Some ap2*)

have $res: res = \text{Inl } [(True, ap2)]$ **using** $res[\text{unfolded } \text{Some}]$ **by** *simp*

```

from large-sort-impl[OF Some finp' simpl'[unfolded simplified-pp-add3]]
  obtain p2 where step: mset3' p'  $\Rightarrow_{ss}$  {#mset3' p2#} and ap2: ap2 =
add3 p2
  and sub: set p2  $\subseteq$  set p' length p2  $\leq$  length p' by auto
from sub simpl'
have simpl2: simplified-pp (add3 p2)
  unfolding simplified-pp-add3 unfolding simplified-pp-def by auto
have vars: tvars-spat (set3 p2)  $\subseteq$  {.. $n+m$ }  $\times$  UNIV  $\wedge$  length p2  $<$  k
  using sub tvars p' by fastforce
from steps' spp-non-det-step[OF step, unfolded mset3-mset3'-set3, OF finp']
have steps: (add-mset (mset3' p) P, {#mset3' p2#} + P)  $\in$  ( $\Rightarrow_{ss}$ )+
  by (rule rtrancl-into-trancl1)
show ?thesis unfolding res ap2 using steps vars simpl2
  by (intro conjI impI exI[of - [(True,p2)]], auto)
next
case clNone: None
note res = res[unfolded clNone option.simps]
show ?thesis
proof (cases search-fun-pp (add3 p'))
  case None
    from res[unfolded None] have res: res = Inr (map (map (map the-Var))
(add3 p^) (is - = Inr ?fvf) by auto
    have id: map (map (map Var)) ?fvf = p' unfolding map-map o-def
    proof (intro map-idI, goal-cases)
      case (1 mp eqc t)
        with search-fun-pp-None[OF None] have is-Var t by force
        thus Var (the-Var (add-sort t)) = t by (cases t, auto)
    qed
    {
      fix mp eqc x  $\iota$  t
      assume *: mp  $\in$  set p' eqc  $\in$  set mp t  $\in$  set eqc (x,  $\iota$ ) = the-Var (add-sort
t)

      with search-fun-pp-None[OF None] have is-Var t by force
      with * have t: t = Var (x, $\iota$ ) by (cases t, auto)
      define terms where terms = concat (concat (add3 p'))
      have tterms: add-sort t  $\in$  set terms unfolding terms-def using * by auto
      define sorts where sorts = remdups (map sort-of terms)
      from t tterms have  $\iota$ sorts:  $\iota \in$  set sorts unfolding sorts-def by force
      from clNone[unfolded large-sort-impl-def Let-def, folded terms-def, folded
sorts-def]
      have extract-option (large-sort-impl-main (add3 p')) sorts = None by auto

      from this[unfolded extract-option-None]  $\iota$ sorts
      have largeNone: large-sort-impl-main (add3 p')  $\iota$  = None by auto

      define find-confl where find-confl = ( $\lambda$ mp.  $\exists$  eqc $\in$ set mp. sort-of (hd (eqc
:: ('f  $\times$  's, nat  $\times$  's) Term.term list)) =  $\iota$ )
      obtain del keep where part: partition find-confl (add3 p') = (del,keep) by
force

```

from *largeNone*[*unfolded large-sort-impl-main-def, folded find-confl-def, unfolded part Let-def split*]
have *fail*: $del = [] \vee \text{length } del \geq \text{cd-sort } \iota \vee (\exists mp \in \text{set } \text{keep}. \exists eq \in \text{set } mp. \exists t \in \text{set } eq. \exists x \in \text{set } (\text{vars-term-list } t). \text{snd } x = \iota)$
by (*auto split: if-splits*)
from *finp'* **have** *finmp*: *finite-constructor-form-mp* (*set2 mp*) **using** *
unfolding *finite-constructor-form-pat-def* **by** *auto*
from * **have** *set eqc* \in *set2 mp* **by** *auto*
from *finmp*[*unfolded finite-constructor-form-mp-def, rule-format, OF this*]
obtain ι' **where** *same*: $\bigwedge t. t \in \text{set } eqc \implies t : \iota' \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS)$ **by** *auto*
from *this*[*OF *(3), unfolded t*] **have** $\iota' = \iota$ **and** *inS*: $\iota \in S$ **by** (*auto simp add: hastype-def restrict-map-def split: if-splits*)
note *same* = *same*[*unfolded this*]
have *find-confl* (*add2 mp*) **unfolding** *find-confl-def*
proof (*intro beXI[of - add1 eqc]*)
show *add1 eqc* \in *set* (*add2 mp*) **using** * **by** *auto*
from * **obtain** *t ts* **where** *eqc*: *eqc* = *t # ts* **by** (*cases eqc, auto*)
from *same*[*of t, unfolded this*] **have** $t : \iota \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS)$ **by** *auto*
hence $t : \iota \text{ in } \mathcal{T}(C, \mathcal{V})$ **by** (*rule typed-restrict-imp-typed*)
with *eqc* **show** *sort-of* (*hd* (*add1 eqc*)) = ι **using** *add-sort[of t ι]* **by** *simp*
qed
with * *part* **have** *add2 mp* \in *set del* **by** *auto*
hence *del* $\neq []$ **by** *auto*
with *fail* **have** *fail*: $\text{length } del \geq \text{cd-sort } \iota \vee (\exists mp \in \text{set } \text{keep}. \exists eq \in \text{set } mp. \exists t \in \text{set } eq. \exists x \in \text{set } (\text{vars-term-list } t). \text{snd } x = \iota)$
(is - \vee ?exists)
by *auto*
hence *card-of-sort C ι* < *k*
proof
assume *len*: $\text{length } del \geq \text{cd-sort } \iota$
from *part* **have** $\text{length } del \leq \text{length } p'$ **by** *auto*
with *tvarsp'* **have** $\text{length } del < k$ **by** *auto*
with *len* **have** $k > \text{cd-sort } \iota$ **by** *auto*
from *this*[*unfolded cd[OF inS]*]
show ?thesis **by** *simp*
next
assume ?exists
then obtain *mp' eq' t' x* **where** **: $mp' \in \text{set } \text{keep } eq' \in \text{set } mp' t' \in \text{set } eq' x \in \text{vars } t' \text{snd } x = \iota$
by *auto*
from *(1) *part* **have** *mp'*: $mp' \in \text{set } (\text{add3 } p')$ **and** $\neg \text{find-confl } mp'$
by *auto*
from *this*(2)[*unfolded find-confl-def*] **
have *neq*: $\text{sort-of } (\text{hd } eq') \neq \iota$ **by** *auto*
from *mp'* **obtain** *mp* **where** *mp*: $mp \in \text{set } p'$ **and** *mp'*: $mp' = \text{add2 } mp$
by *auto*
from *finp'* **have** *finmp*: *finite-constructor-form-mp* (*set2 mp*) **using** *mp*
unfolding *finite-constructor-form-pat-def* **by** *auto*

```

    from **(2)[unfolded mp'] obtain eq where eq: eq ∈ set mp and set eq
    ∈ set2 mp and eq': eq' = add1 eq by auto
    from finmp[unfolded finite-constructor-form-mp-def, rule-format, OF
    this(2)]
    obtain ι' where sort: t ∈ set eq ⇒ t : ι' in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  for t by auto
    from **(3)[unfolded eq'] obtain t where t: t ∈ set eq and t': t' =
    add-sort t by auto
    from t eq mp search-fun-pp-None[OF None] have is-Var t by force
    with **(4,5) t' obtain y where ty: t = Var (y,ι) by (cases t; force)
    from sort[OF t, unfolded ty] have ι' = ι by (auto simp add: hastype-def
    restrict-map-def split: if-splits)
    note sort = sort[unfolded this]
    from t obtain t ts where eq: eq = t # ts by (cases eq, auto)
    hence hd eq' = add-sort t unfolding eq' by auto
    with neq have neq: sort-of (add-sort t) ≠ ι by auto
    from sort[of t, unfolded eq] have t : ι in  $\mathcal{T}(C, \mathcal{V} \mid 'SS)$  by auto
    hence t : ι in  $\mathcal{T}(C, \mathcal{V})$  by (rule typed-restrict-imp-typed)
    from add-sort[OF this] neq have False by auto
    thus ?thesis ..
  qed
}
thus ?thesis unfolding res
  using steps' id simpl' tvarsp' by (fastforce simp: simplified-pp-add3)
next
  case (Some result)
  obtain ap1 amp amp1 aeqc aeqc1 at aeqc2 amp2 ap2 where result: result
  = (ap1, amp, (amp1, aeqc, (aeqc1, at, aeqc2), amp2), ap2)
  by (cases result, auto)
  note res = res[unfolded Some result option.simps split]
  note search = search-fun-pp-Some[OF Some[unfolded result]]
  note ids = search(1-3)
  from ids(1)[unfolded map-eq-append-conv] obtain p1 mp p2 where
  idp: ap1 = add3 p1 amp = add2 mp ap2 = add3 p2 p' = p1 @ mp # p2
  by blast
  from ids(2)[unfolded map-eq-append-conv idp] obtain mp1 eqc mp2 where

  idm: amp1 = add2 mp1 aeqc = map add-sort eqc amp2 = add2 mp2 mp
  = mp1 @ eqc # mp2
  by blast
  from ids(3)[unfolded map-eq-append-conv idm] obtain eqc1 t eqc2 where
  ide: aeqc1 = map add-sort eqc1 at = add-sort t aeqc2 = map add-sort eqc2
  eqc = eqc1 @ t # eqc2
  by blast

  from search have at: is-Fun at at ∈ set aeqc by auto
  from simpl'[unfolded simplified-pp-def] ids
  have simplified-mp amp by auto
  from this[unfolded simplified-mp-def] ids
  have Bex (set aeqc) is-Var by auto

```

hence *find is-Var aeqc* \neq *None* **unfolding** *find-None-iff* **by** *auto*
then obtain *ax* **where** *find: find is-Var aeqc = Some ax* **by** *auto*
from *this[unfolded find-Some-iff]*
have *ax: ax \in set aeqc is-Var ax* **by** *auto*
from *this[unfolded idm]* **obtain** *x* **where** *x: x \in set eqc and is-Var x and*
ax: ax = add-sort x
by *auto*
then obtain *X* **where** *X: x = Var X* **by** (*cases x, auto*)
from *find* **have** *find: the (find is-Var aeqc) = ax* **by** *auto*
define *long* **where** *long = (aeqc1 @ aeqc2) # amp1 @ amp2*
define *ampn* **where** *ampn = (if length-gt-1 (aeqc1 @ aeqc2) then long else*
amp1 @ amp2)
note *res = res[unfolded find Let-def, folded long-def, folded ampn-def]*
from *at ide* **have** *t: is-Fun t* **by** *auto*
let *?cond = amp = [[at, ax]] \vee amp = [[ax, at]]*
show *?thesis*
proof (*cases ?cond*)
case *True*
hence *?cond = True* **by** *auto*
with *res ax X* **have** *res: res = Inl (map (Pair False) (inst-list n X (add3*
p'))) **by** *auto*
from *True* **have** *mp = [[t,x]] \vee mp = [[x,t]]* **unfolding** *idp ax ide* **by** *auto*
hence *{#{#Var X, t#}#} \in # mset3' p'* **unfolding** *idp X* **by** *auto*
from *inst-list[OF this t tvarsp' finp']*
obtain *ps'* **where** *step: mset3' p' \Rightarrow_{ss} mset (map mset3' ps')*
and *inst: map add3 ps' = inst-list n X (add3 p')*
and *vars: Ball (set ps') (λ p'. tvarspat (set3 p') \subseteq {.. $n+m$ } \times UNIV*
 \wedge length p' < k) **by** *blast*
from *steps' spp-non-det-step[OF step, unfolded mset3-mset3'-set3, OF*
finp']
have *steps: (add-mset (mset3' p) P, mset (map mset3' ps') + P) \in (\Rightarrow_{ss})⁺*

by (*rule rtrancl-into-trancl1*)
show *?thesis* **unfolding** *res using steps vars inst[symmetric]*
by (*intro conjI impI exI[of - map (Pair False) ps'], auto simp: o-def*
image-mset.compositionality)
next
case *False*
let *?anew = ((aeqc1 @ aeqc2) # amp1 @ amp2) # ap1 @ ap2*
from *False res*
have *res: res = Inl ((True, ampn # ap1 @ ap2) #*
map (Pair False) (inst-list n (the-Var ax) ([[ax, at]] # ap1 @ ap2)))
by *auto*
from *ide x t X* **have** *x \in set (eqc1 @ eqc2)* **by** *auto*
from *split-list[OF this]* **obtain** *eqc3 eqc4* **where** *eqc12: eqc1 @ eqc2 =*
eqc3 @ x # eqc4 **by** *auto*
from *arg-cong[OF this, of mset] ide*
have *eqc-xt: mset eqc = {#x,t#} + mset (eqc3 @ eqc4)* **and**
x34: add-mset x (mset (eqc3 @ eqc4)) = mset (eqc1 @ eqc2) **by** *auto*

```

have diff1: is-Var x ≠ is-Var t using X t by auto
from False[unfolded idp ax ide] have mp ≠  $[[x,t]] \wedge mp \neq [[t,x]]$  by auto
hence diff2: mset2' mp ≠  $\{\#\{\#x,t\}\#\}$ 
  apply (cases mp)
  apply force
  subgoal for eqc mp'
    apply (cases mp')
    apply (cases eqc)
    apply force
  subgoal for t1 eqc1
    apply (cases eqc1)
    apply force
  subgoal for t2 eqc2
    by (cases eqc2) (auto simp: add-eq-conv-ex)
  done
  apply force
  done
  done
  define np1 where np1 =  $[[x,t]] \# (p1 \text{ @ } p2)$ 
  define np2 where np2 =  $((eqc1 \text{ @ } eqc2) \# (mp1 \text{ @ } mp2)) \# (p1 \text{ @ } p2)$ 
  define np3 where np3 =  $((mp1 \text{ @ } mp2)) \# (p1 \text{ @ } p2)$ 
  have eq: mset2' mp =  $(add\text{-mset } (add\text{-mset } x \ (add\text{-mset } t \ (mset \ (eqc3 \text{ @ } eqc4)))) \ (mset2' \ (mp1 \text{ @ } mp2)))$ 
    unfolding idm by (simp add: eqc-xt)
    with diff2 have  $mset \ (eqc3 \text{ @ } eqc4) \neq \{\#\} \vee mset2' \ (mp1 \text{ @ } mp2) \neq \{\#\}$  by auto
  from spp-split[OF eq diff1 this, unfolded x34, of mset3' (p1 @ p2)]
  have step:  $mset3' \ p' \Rightarrow_{ss} \{\# \ mset3' \ np1, \ mset3' \ np2\}$ 
    by (simp add: idp idm ide np1-def np2-def)
  from steps' spp-non-det-step[OF this, unfolded mset3-mset3'-set3, OF finp', of P]
  have steps:  $(add\text{-mset } (mset \ (map \ (\lambda mpl. \ mset \ (map \ mset \ mpl)) \ p)) \ P, \ \{\# \ mset3' \ np1, \ mset3' \ np2\} + P) \in (\Rightarrow_{ss})^+$ 
    by (rule rtrancl-into-trancl1)

  from tvarsp' have tvars1:  $tvars\text{-spat } (set3 \ np1) \subseteq \{..\lt n\} \times UNIV \wedge \text{length } np1 < k$ 
    using x unfolding np1-def idp idm ide
    by auto
  from tvarsp' have tvars2:  $tvars\text{-spat } (set3 \ np2) \subseteq \{..\lt n\} \times UNIV \wedge \text{length } np2 < k$ 
    unfolding np2-def idp idm ide by auto
  hence tvars2:  $tvars\text{-spat } (set3 \ np2) \subseteq \{..\lt n + m\} \times UNIV \wedge \text{length } np2 < k$  by fastforce
  from spp-step-ms[OF step, unfolded mset3-mset3'-set3, OF finp']
  have fin1: finite-constructor-form-pat (set3 np1)
    and fin2: finite-constructor-form-pat (set3 np2)
    using mset3-mset3'-set3[of np1] mset3-mset3'-set3[of np2] by auto

```

have $\{\#\{\# \text{Var } X, t\#\}\# \} \in \# \text{mset3}' \text{ np1}$ **unfolding** $\text{np1-def } X$ **by** *auto*
from $\text{inst-list}[OF \text{ this } t \text{ tvars1 } \text{fin1}]$
obtain ps' **where** $\text{step: mset3}' \text{ np1} \Rightarrow_{ss} \text{mset} (\text{map } \text{mset3}' \text{ ps}')$
and $\text{inst: map add3 } \text{ps}' = \text{inst-list } n \text{ } X (\text{add3 } \text{np1})$
and $\text{vars: Ball } (\text{set } \text{ps}') (\lambda p'. \text{tvvars-spat } (\text{set3 } p') \subseteq \{..<n+m\} \times \text{UNIV}$
 $\wedge \text{length } p' < k)$ **by** *blast*
from $\text{steps spp-non-det-step}[OF \text{ step, unfolded mset3-mset3}'\text{-set3, } OF \text{ fin1,}$
 $\text{of add-mset } (\text{mset3}' \text{ np2}) \text{ } P]$
have $\text{steps: (add-mset } (\text{mset3}' \text{ } p) \text{ } P, \text{mset } (\text{map } \text{mset3}' \text{ ps}') + \text{add-mset}$
 $(\text{mset3}' \text{ np2}) \text{ } P) \in (\Rightarrow_{ss})^+$
by *simp*
show *?thesis*
proof $(\text{cases length } (\text{aeqc1 } @ \text{aeqc2}) > 1)$
case *True*
hence $\text{ampn: ampn} = (\text{aeqc1 } @ \text{aeqc2}) \# \text{amp1 } @ \text{amp2}$ **unfolding**
 ampn-def long-def **by** *auto*
note $\text{res} = \text{res}[\text{unfolded this}]$
have $\text{simpl-tag}'$: $\text{simpl-tag} (\text{True, ampn } \# \text{ap1 } @ \text{ap2})$
using $\text{simpl}'[\text{unfolded ids}] \text{ at}(1) \text{ True}$ **unfolding** ampn-def long-def
by $(\text{auto simp: simplified-pp-def simplified-mp-def})$
show *?thesis unfolding res*
apply $(\text{intro conjI impI exI[of - (True,np2)] \# \text{map } (\text{Pair False}) \text{ps}'])$
subgoal using $\text{inst}[\text{symmetric}]$ **by** $(\text{auto simp: np2-def idp idm ide}$
 $\text{np1-def } X \text{ ax o-def})$
subgoal using steps **by** $(\text{auto simp: o-def image-mset.compositionality})$
subgoal using vars tvars2 **by** *auto*
subgoal using $\text{simpl-tag}' \text{ ampn}$ **by** *auto*
by *auto*
next
case len: False
hence $\text{ampn: ampn} = \text{amp1 } @ \text{amp2}$ **unfolding** ampn-def **by** *auto*
from len eqc12 **have** $\text{eqc12: eqc1 } @ \text{eqc2} = [x]$ **unfolding** ide **by** $(\text{cases}$
 $\text{eqc1; cases eqc2; auto})$
from $\text{arg-cong}[OF \text{ this, of mset}]$ **have** $\text{single: mset eqc1} + \text{mset eqc2} =$
 $\{\# x \#\}$ **by** *simp*
from eqc12 ide **have** $\text{eqc} = [t,x] \vee \text{eqc} = [x,t]$ **by** $(\text{cases eqc1; auto})$
hence $\text{aeqc} = [at,ax] \vee \text{aeqc} = [ax,at]$ **using** ide ax idm **by** *fastforce*
with False idm idp **have** $\text{ne: mp1} \neq [] \vee \text{mp2} \neq []$ **by** (cases mp1, auto)
have $(\text{add-mset } (\text{mset3}' \text{ np2}) (\text{mset } (\text{map } \text{mset3}' \text{ ps}') + P),$
 $\{\#\text{mset3}' \text{ np3}\#\} + (\text{mset } (\text{map } \text{mset3}' \text{ ps}') + P)) \in \Rightarrow_{ss}$
apply $(\text{rule spp-non-det-step})$
subgoal
apply $(\text{simp add: np2-def np3-def single})$
apply (rule spp-simp)
by $(\text{rule smp-singleton})$
subgoal using $\text{fin2 mset3-mset3}'\text{-set3}[\text{of np2}]$ **by** *auto*
done
with steps
have $\text{steps: (add-mset } (\text{mset3}' \text{ } p) \text{ } P, \text{mset } (\text{map } \text{mset3}' \text{ ps}') + \text{add-mset}$

```

(mset3' np3) P) ∈ (⇒ss)+
  by simp
  have simpl-tag': simpl-tag (True, ampn # ap1 @ ap2)
  using simpl'[unfolding ids] ampn ne idm
  by (simp add: simplified-pp-def simplified-mp-def)
  show ?thesis unfolding res ampn
  apply (intro conjI impI exI[of - (True,np3) # map (Pair False) ps'])
  subgoal using inst[symmetric] by (auto simp: np3-def idp idm ide
np1-def X ax o-def)
  subgoal using steps by (auto simp: o-def image-mset.compositionality)
  subgoal using vars tvars2 np2-def np3-def by auto
  subgoal using simpl-tag' by (auto simp: ampn)
  by auto
qed
qed
qed
qed
qed
qed

```

context

fixes solver :: ((nat×'s) × int)list × - ⇒ bool

assumes fidl: finite-idl-solver solver

begin

lemma pat-complete-via-idl-solver:

assumes 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 small: (∀ x ι. (x,ι) ∈ set (concat (concat cnf))) → card-of-sort C ι < k

and cnf: cnf = map (map snd) fvf

shows pat-complete C (pat-list pp) ↔ ¬ solver (bounds-list (cd-sort o 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)

{

fix v

assume v: v ∈ tvars-pat (pat-list pp)

with wf[unfolding wf-pat-iff] cd

have cd-sort (snd v) = min k (card-of-sort C (snd v)) **by** auto

also have ... = card-of-sort C (snd v) **using** v[folded var-conv] small[rule-format, of fst v snd v]

```

    by auto
    finally have  $cd\text{-sort } (snd\ v) = card\text{-of-sort } C\ (snd\ v)$  .
  } note  $cd\text{-conv} = this$ 

define  $cd :: nat \times 's \Rightarrow nat$  where  $cd = (cd\text{-sort} \circ snd)$ 
define  $S$  where  $S = set\ (concat\ (concat\ cnf))$ 
{
  fix  $v\ vs\ c$ 
  assume  $c \in set\ cnf\ vs \in set\ c\ v \in set\ vs$ 
  hence  $v \in S$  unfolding  $S\text{-def}$  by auto
} note  $in\text{-}S = this$ 
have  $pat\text{-complete } C\ (pat\text{-list } pp) \longleftrightarrow$ 
  ( $\forall \alpha. (\forall v \in S. \alpha\ v < cd\ v) \longrightarrow (\exists c \in set\ cnf. \forall vs \in set\ c. UNIQ\ (\alpha\ 'set\ vs)))$ )
  by ( $unfold\ S\text{-def}\ pat\text{-complete-via-cnf}[OF\ fvf\ pp\ dist\ cnf]\ var\text{-conv},\ simp\ add:$ 
 $cd\text{-conv}\ cd\text{-def}$ )
  also have  $\dots \longleftrightarrow \neg (\exists \alpha. (\forall v \in S. \alpha\ v < cd\ v) \wedge (\forall c \in set\ cnf. \exists vs \in set\ c. \neg$ 
 $UNIQ\ (\alpha\ 'set\ vs)))$  (is -  $\longleftrightarrow \neg\ ?f$ ) by blast
  also have  $?f \longleftrightarrow (\exists \alpha. (\forall v \in S. \alpha\ v < cd\ v) \wedge (\forall c \in set\ cnf. \exists vs \in set\ c. \exists v \in set$ 
 $vs. \exists w \in set\ vs. \alpha\ v \neq \alpha\ w))$  (is -  $\longleftrightarrow (\exists \alpha. ?fN\ \alpha)$ )
    unfolding  $non\text{-uniq-image-diff}$  ..
  also have  $\dots \longleftrightarrow (\exists \alpha. (\forall v \in S. 0 \leq \alpha\ v \wedge \alpha\ v < int\ (cd\ v)) \wedge (\forall c \in set\ cnf.$ 
 $\exists vs \in set\ c. \exists v \in set\ vs. \exists w \in set\ vs. \alpha\ v \neq \alpha\ w))$  (is -  $\longleftrightarrow (\exists \alpha. ?fZ\ \alpha)$ )
  proof
    assume  $\exists \alpha. ?fN\ \alpha$ 
    then obtain  $\alpha$  where  $?fN\ \alpha$  by blast
    hence  $?fZ\ (int\ o\ \alpha)$  unfolding  $o\text{-def}$  by auto
    thus  $\exists \alpha. ?fZ\ \alpha$  by blast
  next
    assume  $\exists \alpha. ?fZ\ \alpha$ 
    then obtain  $\alpha$  where  $alpha: ?fZ\ \alpha$  by blast
    have  $?fN\ (nat\ o\ \alpha)$  unfolding  $o\text{-def}$ 
    proof (intro  $conjI\ ballI$ )
      show  $v \in S \implies nat\ (\alpha\ v) < cd\ v$  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  $in\text{-}S[OF\ c\ vs]\ v\ w$  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  $\dots \longleftrightarrow (\exists \alpha. (\forall v \in S. 0 \leq \alpha\ v \wedge \alpha\ v \leq int\ (cd\ v) - 1) \wedge (\forall c \in set$ 
 $cnf. \exists vs \in set\ c. \exists v \in set\ vs. \exists w \in set\ vs. \alpha\ v \neq \alpha\ w))$ 
  by auto

```

also have ... = $(\exists \alpha. (\forall (v, b) \in \text{set } (\text{bounds-list } \text{cd } \text{cnf}). 0 \leq \alpha v \wedge \alpha v \leq b) \wedge (\forall c \in \text{set } (\text{dist-pairs-list } \text{cnf}). \exists (v, w) \in \text{set } c. \alpha v \neq \alpha w))$
unfolding *bounds-list-def Let-def S-def[symmetric] set-map set-remdups*
proof (*intro arg-cong[of - - Ex] ext arg-cong2[of - - - (\wedge)], force*)
fix $\alpha :: - \Rightarrow \text{int}$
show $(\forall c \in \text{set } \text{cnf}. \exists vs \in \text{set } c. \exists v \in \text{set } vs. \exists w \in \text{set } vs. \alpha v \neq \alpha w) = (\forall c \in \text{set } (\text{dist-pairs-list } \text{cnf}). \exists (v, w) \in \text{set } c. \alpha v \neq \alpha w)$
unfolding *diff-pairs-of-list dist-pairs-list-def set-map image-comp set-concat o-def*
by force
qed
also have ... = *fidl-solvable (bounds-list cd cnf, dist-pairs-list cnf)*
unfolding *fidl-solvable-def split ..*
also have ... = *solver (bounds-list cd cnf, dist-pairs-list cnf)*
proof (*rule sym, rule fidl[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 $(x, y) \in \text{set } (\text{concat } (\text{dist-pairs-list } \text{cnf})) \implies z \in \{x, y\} \implies z \in \text{fst 'set } (\text{bounds-list } \text{cd } \text{cnf}) \text{ for } x y z$
unfolding *dist-pairs-list-def bounds-list-def List.maps-eq 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 $\bigwedge v w b1 b2. (v, b1) \in \text{set } (\text{bounds-list } \text{cd } \text{cnf}) \implies (w, b2) \in \text{set } (\text{bounds-list } \text{cd } \text{cnf}) \implies \text{snd } v = \text{snd } w \implies b1 = b2$
unfolding *bounds-list-def Let-def* **by** (*auto simp: cd-def*)
{
fix $v b$
assume $(v, b) \in \text{set } (\text{bounds-list } \text{cd } \text{cnf})$
from *this[unfolded bounds-list-def]*
have $v: v \in \text{tvars-pat } (\text{pat-list } \text{pp})$ **and** $b: b = \text{int } (\text{cd } v) - 1$ **by** (*auto simp flip: var-conv*)
from *cd-conv[OF v] b* **have** $b: b = \text{int } (\text{card-of-sort } C (\text{snd } v)) - 1$ **by** (*auto simp: cd-def*)
from *wf[unfolded wf-pat-iff, rule-format, OF v]*
have $vS: \text{snd } v \in ?S$ **by** *auto*
from *not-empty-sort[OF this]*
have $nE: \neg \text{empty-sort } C (\text{snd } v)$.
from *v[unfolded tvars-pat-def tvars-match-def]*
obtain $mp \ t \ l$ **where** $mp: mp \in \text{pat-list } \text{pp}$ **and** $tl: (t, l) \in mp$ **and** $vt: v \in \text{vars } t$ **by** *auto*
from *fuf[unfolded finite-var-form-pat-def] mp* **have** $mp: \text{finite-var-form-match } C \ mp$ **by** *auto*
note $mp = mp[unfolded \text{finite-var-form-match-def}]$
from *mp[unfolded var-form-match-def] tl* **obtain** x **where** $t: t = \text{Var } x$ **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, 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

lemma fvf-solver: assumes tfvf: tfvf = map (map (map (Var :: nat × 's ⇒ ('f,
nat × 's)term))) fvf
  and small: (∀ x ι. (x,ι) ∈ set (concat (concat fvf))) → card-of-sort C ι < k)
  and fin: finite-constructor-form-pat (set3 tfvf)
  and simpl: simplified-pp tfvf
shows fvf-solver solver fvf = simple-pat-complete C SS (set3 tfvf)
proof -
{
  fix eqc :: (nat × 's) list
  have set (zip eqc (tl eqc)) = (λ i. (eqc ! i, eqc ! (Suc i))) ‘ {..< length eqc -
1}
  unfolding set-zip by (cases eqc, auto)
} note set-zip-etc = this
let ?Var = Var :: nat × 's ⇒ ('f, nat × 's)term
from fin[unfolded finite-constructor-form-pat-def tfvf finite-constructor-form-mp-def]
have fin: eqc ∈ set mp ⇒ mp ∈ set fvf ⇒ eqc ≠ [] ∧ (∃ ι. finite-sort C ι ∧

```

```

( $\forall x \in \text{set eqc. } x : \iota \text{ in } \mathcal{V} \mid 'SS)$ )
  for eqc mp by auto
  {
    fix eqc mp
    assume mp  $\in$  set fvf eqc  $\in$  set mp
    with simpl[unfolded tfvf simplified-pp-def simplified-mp-def] have distinct (map
?Var eqc) by auto
    hence distinct eqc unfolding distinct-map by auto
  } note dist-egc = this
  define fvf' where fvf' = map (  $\lambda$  mp. zip [0.. $\text{length mp}$ ] mp) fvf
  have fvf': fvf = map (map snd) fvf' unfolding fvf'-def map-map o-def
    by (rule sym, rule map-idI, rule nth-equalityI, auto)
  have dist1: Ball (set fvf') (distinct  $\circ$  map fst) unfolding fvf'-def
    by auto
  have dist2: Ball (set (concat fvf')) (distinct  $\circ$  snd)
    unfolding fvf'-def
    by (auto simp: set-zip intro: dist-egc)
  define pp :: ((f, nat  $\times$  's) term  $\times$  (f, nat) term) list list where pp = pat-of-var-form-list
  fvf'
  {
    fix mp i
    assume mp: mp  $\in$  set fvf i < length mp
    hence mp ! i  $\in$  set mp by auto
    from dist-egc[OF mp(1) this] fin[OF this mp(1)] obtain  $\iota$ 
      where dist: distinct (mp ! i) finite-sort C  $\iota$  ( $\forall x \in \text{set (mp ! i). } x : \iota \text{ in } \mathcal{V} \mid '
SS)$  by auto
    hence  $\forall a b. (a,b) \in \text{set (mp ! i)} \longrightarrow b = \iota \wedge b \in S$  unfolding hastype-def
  restrict-map-def
    by (auto split: if-splits)
    with dist(1-2) have distinct (mp ! i)  $\exists \iota. \text{finite-sort } C \iota \wedge (\forall x s. (x,s) \in \text{set}
(mp ! i) \longrightarrow s = \iota \wedge s \in S)$ 
      by blast+
  } note mpi = this

  have wf-pat: wf-pat (pat-list pp) unfolding pat-list-def pp-def pat-of-var-form-list-def
  using mpi(2)
    by (force simp: wf-pat-def wf-match-def match-of-var-form-list-def tvars-match-def
  fvf'-def set-zip)

  have var-form: finite-var-form-pat C (pat-list pp) unfolding pp-def fvf'-def
    apply (auto simp: pat-list-def pat-of-var-form-list-def match-of-var-form-list-def
  finite-var-form-pat-def
      finite-var-form-match-def var-form-match-def set-zip)
    subgoal for mp x s y s' i using mpi(2)[of mp i] by blast
    subgoal for mp x s i using mpi(2)[of mp i] by blast
  done
  from pat-complete-via-idl-solver[OF var-form wf-pat pp-def dist1 dist2 small fvf']
  have pat-complete C (pat-list pp) = fvf-solver solver fvf unfolding fvf-solver-def
  .

```

```

also have pat-complete  $C$  (pat-list pp) = (( $\forall f :_s \mathcal{V} \mid ' SS \rightarrow \mathcal{T}(C)$ ). Bex (pat-list
pp) (match-complete-wrt f)))
  unfolding pat-complete-def
  proof ((standard; intro allI impI), goal-cases)
  case (1 f)
  have tvars-pat (pat-list pp)  $\subseteq SS$ 
  using wf-pat unfolding wf-pat-def wf-match-def tvars-match-def tvars-pat-def
by force
  with 1 have f:  $f :_s \mathcal{V} \mid ' tvars-pat$  (pat-list pp)  $\rightarrow \mathcal{T}(C)$ 
  by (auto simp: sorted-map-def tvars-pat-def restrict-map-def hastype-def)
  with 1 show Bex (pat-list pp) (match-complete-wrt f) by auto
next
  case (2 f)
  define g where  $g x =$  (if  $x \in tvars-pat$  (pat-list pp) then  $f x$  else  $\sigma g' x$ ) for x
  have g:  $g :_s \mathcal{V} \mid ' SS \rightarrow \mathcal{T}(C)$  using  $\sigma g' 2(2)$ 
  by (auto simp: g-def sorted-map-def tvars-pat-def restrict-map-def hastype-def
split: if-splits)
  with 2(1) obtain mp where mp:  $mp \in$  pat-list pp and match: match-complete-wrt
g mp by auto
  have match-complete-wrt g mp = match-complete-wrt f mp unfolding match-complete-wrt-def
  apply (intro ex-cong1 ball-cong refl, clarsimp)
  subgoal for mu s t
  by (subst term-subst-eq[of s f g], insert mp, auto simp: g-def tvars-pat-def
tvars-match-def)
  done
  with match show Bex (pat-list pp) (match-complete-wrt f) using mp by auto
qed
also have ... = simple-pat-complete  $C SS$  (set3 tfvf)
  unfolding simple-pat-complete-def pat-list-def tfvf set-map o-def pp-def fvf'
  pat-of-var-form-list-def image-comp bex-simps
proof (intro all-cong bex-cong refl)
  fix f mp
  assume f :_s ( $\lambda x$ . Some (snd x))  $\mid ' SS \rightarrow \mathcal{T}(C)$ 
  assume mp  $\in$  set fvf'
  with dist1 have dist1: distinct (map fst mp) by auto
  show match-complete-wrt f (set (match-of-var-form-list mp)) =
  simple-match-complete-wrt f {Var ' set (snd eqc)  $\mid$ . eqc  $\in$  set mp}
  unfolding match-complete-wrt-def simple-match-complete-wrt-def ball-simps
proof
  assume *:  $\forall eqc \in$  set mp. UNIQ-subst f (Var ' set (snd eqc))
  define  $\mu$  where  $\mu n =$  the-elem (Var ' set (the (map-of mp n))  $\cdot_{set}$  f) for n
  show  $\exists \mu$ .  $\forall (t, l) \in$  set (match-of-var-form-list mp).  $t \cdot f = l \cdot \mu$ 
  proof (intro exI[of -  $\mu$ ], clarsimp simp: match-of-var-form-list-def)
  fix n eqc x s
  assume eqc: (n,eqc)  $\in$  set mp and x: (x,s)  $\in$  set eqc
  from * eqc have uniq: UNIQ-subst f (Var ' set eqc) by auto
  from eqc dist1 have the: the (map-of mp n) = eqc by simp
  hence  $\mu$ :  $\mu n =$  the-elem (Var ' set eqc  $\cdot_{set}$  f) unfolding  $\mu$ -def by simp
  from Uniq-eq-the-elem[OF uniq[unfolded UNIQ-subst-def], folded this, of f

```

```

(x, s)]
  show f (x,s) = μ n using x by force
qed
next
  assume ∃ μ. ∀ (t, l) ∈ set (match-of-var-form-list mp). t · f = l · μ
  then obtain μ where μ: ∧ t l. (t, l) ∈ set (match-of-var-form-list mp) ⇒
t · f = l · μ by auto
  show ∀ eqc ∈ set mp. UNIQ-subst f (Var ‘ set (snd eqc))
  proof (intro ballI, clarsimp)
    fix n eqc
    assume (n, eqc) ∈ set mp
    hence Var ‘ set eqc × {Var n} ⊆ set (match-of-var-form-list mp)
      unfolding match-of-var-form-list-def by auto
    from μ[OF set-mp[OF this]] have μ: x ∈ set eqc ⇒ f x = μ n for x by
fastforce
    thus UNIQ-subst f (Var ‘ set eqc) unfolding UNIQ-subst-alt-def by auto
  qed
qed
qed
finally show ?thesis by simp
qed

```

```

lemma fcf-solver-loop: assumes vars: Ball (snd ‘ set P) (λ p. tvars-spat (set3 p)
⊆ {.. $n$ } × UNIV ∧ length p < k)
  and prob: spp-det-prob (mset4' P)
  and tags: Ball (set (add4 P)) simpl-tag
shows fcf-solver-loop solver n (add4 P) = spp-pat-complete (mset4' P)
  using vars prob tags
proof (induct P arbitrary: n rule: SN-induct[OF SN-inv-image[of - mset4', OF
SN-imp-SN-trancl[OF SN-spp-det-step]]])
  case (1 P n)
  note vars = 1(2)
  note prob = 1(3)
  note tags = 1(4)
  note IH = 1(1)
  note res = fcf-solver-loop.simps[of solver n add4 P]
  show ?case
  proof (cases P)
    case Nil
    thus ?thesis unfolding res by (auto simp: spp-pat-complete-def)
  next
  case (Cons tp ps)
  then obtain tag p where Cons: P = (tag,p) # ps by (cases tp, auto)
  hence P: add4 P = (tag,add3 p) # add4 ps by auto
  note res = res[unfolded P list.simps, folded P]
  from prob[unfolded Cons spp-det-prob-def]
  have finp: finite-constructor-form-pat (set3 p)
    using mset3-mset3'-set3[of p] by auto
  from vars[unfolded Cons]

```

```

have varsp: tvars-spat (set3 p)  $\subseteq$  {.. $n$ }  $\times$  UNIV  $\wedge$  length p < k by auto
from tags[unfolded P] have tag: simpl-tag (tag, add3 p)
  and tags-ps: Ball (set (add4 ps)) simpl-tag by auto
show ?thesis
proof (cases full-step n (tag, add3 p))
  case (Inl aps1)
    from full-step(1)[OF varsp finp tag refl Inl, of mset4' ps] Cons
    obtain ps1 where aps1: aps1 = add4 ps1 and
      steps: (mset4' P, mset4' ps1 + mset4' ps)  $\in$  ( $\cong_{ss}$ )+ and
      varsp: Ball (snd ' set ps1) ( $\lambda$  p'. tvars-spat (set3 p')  $\subseteq$  {.. $n + m$ }  $\times$ 
UNIV  $\wedge$  length p' < k) and
      tags: Ball (set aps1) simpl-tag
    by auto
    from res[unfolded Inl sum.simps aps1]
    have res: fcf-solver-loop solver n (add4 P) = fcf-solver-loop solver (n + m)
(add4 (ps1 @ ps))
    by simp
    from varsp: vars
    have vars:  $\forall p \in$  snd ' set (ps1 @ ps). tvars-spat (set3 p)  $\subseteq$  {.. $n+m$ }  $\times$ 
UNIV  $\wedge$  length p < k
    unfolding Cons by fastforce
    from tags[unfolded aps1] tags-ps
    have tags: Ball (set (add4 (ps1 @ ps))) simpl-tag by auto
    have (P, ps1 @ ps)  $\in$  inv-image ( $\cong_{ss}$ )+ mset4' using steps by auto
    note IH = IH[OF this vars - tags]
    from steps have steps: (mset4' P, mset4' (ps1 @ ps))  $\in$  ( $\cong_{ss}$ )* by simp
    from spp-det-steps-ms[OF this] prob
    have prob: spp-det-prob (mset4' (ps1 @ ps))
      and sound: spp-pat-complete (mset4' P) = spp-pat-complete (mset4' (ps1
@ ps)) by blast+
    show ?thesis
    unfolding res
    unfolding IH[OF prob]
    unfolding sound ..
  next
  case (Inr fvf)
  let ?fvf = map (map (map (Var :: nat  $\times$  's  $\Rightarrow$  ('f, nat  $\times$  's)term))) fvf
  note res = res[unfolded Inr sum.simps]
  let ?new = add-mset (mset3' ?fvf) (mset4' ps)
  from full-step(2)[OF varsp finp tag refl Inr, of mset4' ps] Cons
  have steps: (mset4' P, ?new)  $\in$  ( $\cong_{ss}$ )*
  and simpl: simplified-pp ?fvf
  and small: ( $\forall x \iota. (x, \iota) \in$  set (concat (concat fvf))  $\longrightarrow$  card-of-sort C  $\iota$  <
k) by auto
  from spp-det-steps-ms[OF steps] prob have spp-det-prob ?new
  by (auto simp: o-def image-mset.compositionality)
  from this[unfolded spp-det-prob-def] have finite-constructor-form-pat (mset3
(mset3' ?fvf)) by auto
  hence fin: finite-constructor-form-pat (set3 ?fvf) unfolding mset3-mset3'-set3

```

```

have solver: fvf-solver solver fvf = simple-pat-complete C SS (set3 ?fvf)
  by (rule fvf-solver[OF refl small fin simpl])
have fvf: ( $\bigwedge t. t \in \# \sum \# (\text{image-mset } \sum \# (\text{mset3}' ?fvf)) \implies \text{is-Var } t$ )
  by auto
show ?thesis
proof (cases fvf-solver solver fvf)
  case True
    from spp-fvf-succ[of mset3' ?fvf, unfolded mset3-mset3'-set3, OF fvf
True[unfolded solver]]
    have step: (?new, mset4' ps)  $\in \equiv_{ss}$  by auto
    with steps have steps: (mset4' P, mset4' ps)  $\in (\equiv_{ss})^+$  by auto
    from res True have res: fcf-solver-loop solver n (add4 P) = fcf-solver-loop
solver n (add4 ps) by auto
    from vars have vars:  $\forall p \in \text{snd } \text{' set ps. } \text{tvars-spat (set3 p)} \subseteq \{..<n\} \times \text{UNIV}$ 
 $\wedge \text{length } p < k$ 
      unfolding Cons by fastforce
      have (P, ps)  $\in \text{inv-image } (\equiv_{ss}^+) \text{ mset4' using steps by auto}$ 
      note IH = IH[OF this vars - tags-ps]
      from steps have steps: (mset4' P, mset4' ps)  $\in (\equiv_{ss})^* ..$ 
      from spp-det-steps-ms[OF this] prob
      have prob: spp-det-prob (mset4' ps)
        and sound: spp-pat-complete (mset4' P) = spp-pat-complete (mset4' ps)
        by (auto simp: o-def image-mset.compositionality)
      show ?thesis
        unfolding res
        unfolding IH[OF prob]
        unfolding sound ..
    next
      case False
      with res have res: fcf-solver-loop solver n (add4 P) = False by auto
      from False[unfolded solver]
      have  $\neg$  simple-pat-complete C SS (set3 ?fvf) by auto
      hence  $\neg$  spp-pat-complete ?new unfolding spp-pat-complete-def o-def
        using mset3-mset3'-set3[of ?fvf] by auto
      with spp-det-steps-ms[OF steps]
      show ?thesis unfolding res unfolding spp-pat-complete-def by auto
    qed
  qed
qed
qed

```

```

lemma fcf-solver-alg: assumes vars: tvars-spat (set3 p)  $\subseteq \{..<n\} \times \text{UNIV}$ 
and k: length p < k
and fin: finite-constructor-form-pat (set3 p)
shows fcf-solver-alg solver n p = simple-pat-complete C SS (set3 p)
proof -
  have fcf-solver-alg solver n p = fcf-solver-loop solver n (add4 [(False,p)])
  unfolding fcf-solver-alg-def by simp

```

```

also have ... = spp-pat-complete ({#mset3' p#}) using mset3-mset3'-set3[of
p]
by (subst fcf-solver-loop, insert vars fin k, auto)
also have ... = simple-pat-complete C SS (set3 p) using mset3-mset3'-set3[of
p]
unfolding spp-pat-complete-def by auto
finally show ?thesis .
qed

```

```

lemma fcf-solver-alg': fcf-solver k (fcf-solver-alg solver)
unfolding fcf-solver-def using fcf-solver-alg by auto
end
end

```

```

context pattern-completeness-context
begin

```

```

lemmas fcf-solver-code-eqns =
fcf-solver-alg-def
fcf-solver-loop.simps
fuf-solver-def
full-step-def
simplify-tp.simps
simplify-pp.simps
simplify-mp-def
simplify-mp-main.simps
add-sort.simps
inst-list-def[unfolded s $\tau$ s-list-def s $\tau$ c-def map-map]
large-sort-impl-def
large-sort-impl-main-def

```

```

end

```

```

declare pattern-completeness-context.fcf-solver-code-eqns[code]

```

```

end

```

```

theory Finite-IDL-Solver

```

```

imports

```

```

HOL-Library.RBT-Mapping
HOL-Library.List-Lexorder
HOL-Library.Product-Lexorder
Finite-IDL-Solver-Interface
Singleton-List
Polynomial-Factorization.Missing-List

```

```

begin

```

Delete all variables with (a sort that has) an upper bound of 0; if some the clauses becomes empty, return a trivial unsat-problem.

definition *delete-trivial-sorts* :: ('v,'s)fidl-input ⇒ ('v,'s)fidl-input option **where**
delete-trivial-sorts = (λ (bnds, diffs).
 case partition ((=) 0 o snd) bnds of
 ([,-] ⇒ Some (bnds, diffs)
 | (triv,non-triv) ⇒ let triv-sorts = set (map (snd o fst) triv);
 newdiffs = map (filter (λ vw. snd (fst vw) ∉ triv-sorts)) diffs
 in if [] ∈ set newdiffs then None else Some (non-triv, newdiffs))

lemma *delete-trivial-sorts: assumes inp: fidl-input input*
and *del: delete-trivial-sorts input = ooutput*
shows (ooutput = None → ¬ fidl-solvable input) ∧ (ooutput = Some output →
 fidl-input output ∧ fidl-solvable input = fidl-solvable output)
proof –
obtain *bnds diffs where input: input = (bnds,diffs) by force*
obtain *triv non-triv where part: partition ((=) 0 o snd) bnds = (triv,non-triv)*
(is partition ?f - = -) by force
define *f where f = ?f*
note *del = del[unfolded delete-trivial-sorts-def input split part]*
show *?thesis*
proof (*cases triv*)
 case *Nil*
thus *?thesis using inp del input by auto*
next
 case *Cons*
from *part[unfolded partition-filter-conv, folded f-def]*
have *triv: triv = filter f bnds and non-triv: non-triv = filter (Not o f) bnds*
by auto
define *tsorts where tsorts = set (map (snd o fst) triv)*
define *newdiffs where newdiffs = map (filter (λvs. snd (fst vs) ∉ tsorts)) diffs*

let *?out = (non-triv, newdiffs)*
note *inp = inp[unfolded input fidl-input-def split]*
from *inp have dist: distinct (map fst bnds) by blast*
have *out-conds: fidl-input ?out*
unfolding *fidl-input-def split*
proof (*intro conjI allI impI*)
show *distinct (map fst non-triv) unfolding non-triv by (rule distinct-map-filter[OF*
dist])
 {
fix *v w*
assume *(v, w) ∈ set (concat newdiffs)*
from *this[unfolded newdiffs-def]*
have *vw: (v, w) ∈ set (concat diffs) and snd: snd v ∉ tsorts by auto*
from *vw inp show eq: v ≠ w by blast*
from *vw inp show eq: snd v = snd w by blast*
from *vw inp have {v,w} ⊆ fst ' set bnds by blast*
with *snd eq have {v,w} ⊆ fst ' set non-triv unfolding non-triv triv tsorts-def*
by force
thus *u ∈ {v, w} ⇒ u ∈ fst ' set non-triv for u by auto*

```

}
qed (insert inp, auto simp: non-triv)
let ?sat1 = λ bnds α. (∀ (v, b) ∈ set bnds. 0 ≤ α v ∧ α v ≤ b)
let ?sat2 = λ diffs α. (∀ c ∈ set diffs. ∃ (v, w) ∈ set c. α v ≠ α w)
let ?sat = λ bnds diffs α. ?sat1 bnds α ∧ ?sat2 diffs α
have set-bnds: set bnds = set triv ∪ set non-triv using part by fastforce
have main: fidl-solvable input = fidl-solvable ?out unfolding input
proof
  assume fidl-solvable (non-triv, newdiffs)
  from this[unfolded fidl-solvable-def split] obtain α
    where sat1: ?sat1 non-triv α and sat2: ?sat2 newdiffs α by blast
  define β where β v = (if snd v ∈ tsorts then 0 else α v) for v
  have αβ: ?sat2 newdiffs α = ?sat2 newdiffs β
  proof (intro ball-cong[OF refl] bex-cong[OF refl], clarsimp)
    fix vsf v1 s1 v2 s2
    assume *: vsf ∈ set newdiffs ((v1,s1),(v2,s2)) ∈ set vsf
    from *(1)[unfolded newdiffs-def]
    obtain vs where vs: vs ∈ set diffs and vsf: vsf = filter (λvs. snd (fst vs)
  ∉ tsorts) vs by auto
    from *(2)[unfolded vsf]
    have vw: ((v1,s1),(v2,s2)) ∈ set vs and s1: s1 ∉ tsorts by auto
    from inp vw vs have s1 = s2 by fastforce
    with s1 show (α (v1,s1) = α (v2,s2)) = (β (v1,s1) = β (v2,s2)) unfolding
  β-def by auto
  qed
  have ?sat1 bnds β unfolding set-bnds using sat1
    unfolding β-def triv non-triv tsorts-def f-def by force
  moreover have ?sat2 diffs β using sat2[unfolded αβ] unfolding newdiffs-def
  by auto
  ultimately show fidl-solvable (bnds, diffs) unfolding fidl-solvable-def by
  auto
next
  assume fidl-solvable (bnds, diffs)
  from this[unfolded fidl-solvable-def split] obtain α
    where sat1: ?sat1 bnds α and sat2: ?sat2 diffs α by blast
  from sat1 have ?sat1 non-triv α unfolding non-triv by auto
  moreover have ?sat2 newdiffs α
  proof
    fix vsf
    assume vsf ∈ set newdiffs
    from this[unfolded newdiffs-def]
    obtain vs where vs: vs ∈ set diffs and vsf: vsf = filter (λvs. snd (fst vs)
  ∉ tsorts) vs by auto
    from sat2[rule-format, OF vs] obtain v w where vw: (v,w) ∈ set vs and
  diff: α v ≠ α w by auto
    from vs vw have vw': (v,w) ∈ set (concat diffs) by auto
    from inp vw' have vw-sort: snd v = snd w by blast
    from inp vw' have vw-bnds: {v,w} ⊆ fst ' set bnds by blast
    then obtain b bw where {(v,b),(w,bw)} ⊆ set bnds by auto

```

```

with inp vw-sort have bnds: {(v,b),(w,b)} ⊆ set bnds by (metis insert-subset)
with sat1 have  $0 \leq \alpha v \alpha v \leq b \ 0 \leq \alpha w \alpha w \leq b$  by auto
with diff have b0: b ≠ 0 by auto
have snd v ∉ tsorts
proof
  assume snd v ∈ tsorts
  from this[unfolded tsorts-def triv f-def]
  obtain u where u0: (u,0) ∈ set bnds and same-sort: snd u = snd v by
auto
  from bnds inp u0 same-sort have b = 0 by blast
  with b0 show False by auto
qed
with vw have  $(v,w) \in set vsf$  unfolding vsf by auto
with diff show  $\exists (v, w) \in set vsf. \alpha v \neq \alpha w$  by auto
qed
ultimately show fidl-solvable (non-triv, newdiffs)
  by (auto simp: fidl-solvable-def)
qed
have outp: ooutput = (if [] ∈ set newdiffs then None else Some ?out)
  using del by (auto simp: Cons tsorts-def newdiffs-def Let-def)
show ?thesis
proof (cases [] ∈ set newdiffs)
  case False
  thus ?thesis using outp main out-conds by auto
next
  case True
  from split-list[OF this]
  show ?thesis using outp unfolding main
  by (auto simp: fidl-input-def fidl-solvable-def)
qed
qed
qed

```

fun *assign-by-sort* :: $('s, (int \times ('v \times int)list))$ *mapping* $\Rightarrow ((v \times 's) \times int)$ *list*
 $\Rightarrow ('s, int \times ('v \times int)list)$ *mapping* **where**
assign-by-sort *m [] = m*
| *assign-by-sort* *m (((v,s),b) # bnds) = (case Mapping.lookup m s of*
None \Rightarrow *assign-by-sort (Mapping.update s (b,[(v,b)]) m) bnds*
| *Some (b,vs)* \Rightarrow *assign-by-sort (Mapping.update s (b - 1, (v, b - 1) # vs) m)*
bnds)

lemma *assign-by-sort-computation: fixes bnds :: ((v × 's) × int) list and s :: 's*
assumes *filt: filt = filter ((=) s ∘ snd ∘ fst) bnds*
shows *Mapping.lookup (assign-by-sort Mapping.empty bnds) s = (if filt = [] then*
None
else Some
(snd (hd filt) - int (length filt) + 1,
*rev (map (λi. (fst (fst (filt ! i)), snd (hd filt) - int i)) [0..*length filt*]))*
proof -

```

let ?f = λ m ((v :: 'v, s :: 's), b :: int). case m of None ⇒ Some (b :: int, [(v, b)])
| Some (b :: int, vs) ⇒ Some (b - 1, (v, b - 1) # vs)
let ?filt = filt
define f where f = ?f
define fi :: (('v × 's) × int) ⇒ bool where fi = (((=) s) o snd o fst)
{
  fix a bnds' m
  have Mapping.lookup m s = foldl f a (filter fi bnds')
  ⇒ Mapping.lookup (assign-by-sort m bnds) s =
  foldl f a (filter fi (bnds' @ bnds))
  proof (induct bnds arbitrary: m bnds')
  case (Cons entry bnds m bnds')
  obtain v s' b where entry: entry = ((v, s'), b) by (cases entry) auto
  show ?case
  proof (cases s' = s)
  case False
  then obtain m' where
    id: assign-by-sort m (entry # bnds) = assign-by-sort m' bnds and m':
    Mapping.lookup m' s = Mapping.lookup m s
    unfolding entry by (cases Mapping.lookup m s', auto)
    from False have False: ¬ fi entry unfolding fi-def entry by auto
    show ?thesis unfolding id
    by (subst Cons(1), rule Cons(2)[folded m'], insert False, auto simp: entry)

  next
  case True
  obtain m' where
    id: assign-by-sort m (entry # bnds) = assign-by-sort m' bnds
    and m': Mapping.lookup m' s = f (Mapping.lookup m s) entry
    unfolding entry True by (cases Mapping.lookup m s, auto simp: f-def)
    from True have fi: fi entry unfolding entry fi-def by auto
    have filt: filter fi ((entry # bnds) @ bnds') = entry # filter fi (bnds @ bnds')

    unfolding entry True fi-def by auto
    show ?thesis unfolding id filt
    by (subst Cons(1)[where bnds' = bnds' @ [entry]], subst m'[unfolded
    Cons(2)], insert fi, auto)
  qed
  qed auto
}
from this[of Mapping.empty None Nil]
have impl: Mapping.lookup (assign-by-sort Mapping.empty bnds) s = foldl f None
(filter fi bnds)
by auto
define filt where filt = filter fi bnds
{
  assume filt ≠ []
  then obtain x0 b0 s' bnds'
  where filt: filt = ((x0, s'), b0) # bnds' by (cases filt, auto)
}

```

```

have fold: foldl f (Some (b,xs)) bnds' =
  Some (b - int (length bnds'),
    rev (map2 (λi v. (fst (fst v), b - i)) [1.. int (length bnds')] bnds') @ xs)
for b xs
proof (induct bnds' arbitrary: b xs)
  case (Cons entry bnds)
  obtain x s b' where entry: entry = ((x,s),b') by (cases entry) auto
  have f-entry: f (Some (b, xs)) entry = Some (b - 1, (x, b - 1) # xs)
    unfolding f-def entry by simp
  have list: [1..int (length bnds + Suc 0)] = 1 # map ((+) 1) [1..int (length
bnds)]
    apply (intro nth-equalityI, force)
    subgoal for i by (cases i, auto)
    done
  have map-eq: map2 (λx y. (fst (fst y), b - 1 - x)) [1..int (length bnds)] bnds
=
  map2 (λx y. (fst (fst y), b - x)) (map ((+) 1) [1..int (length bnds)]) bnds
  by (intro nth-equalityI, auto)
  show ?case unfolding foldl.simps list.size f-entry
    unfolding Cons option.simps prod.simps unfolding list
    by (intro conjI, force, insert map-eq, simp add: entry)
qed auto
have Mapping.lookup (assign-by-sort Mapping.empty bnds) s = foldl f (Some
(b0,[x0,b0])) bnds'
  unfolding impl filt-def[symmetric] filt by (simp add: f-def)
also have ... = Some
  (b0 - int (length filt) + 1,
    rev (map (λ i. (fst (fst (filt ! i)), b0 - int i)) [0 ..< length filt]))
  unfolding fold option.simps prod.simps rev.simps(2)[symmetric]
proof (intro conjI arg-cong[of - - rev])
  show (x0, b0) # map2 (λx y. (fst (fst y), b0 - x)) [1..int (length bnds')]
bnds' =
  map (λi. (fst (fst (filt ! i)), b0 - int i)) [0..<length filt]
  unfolding filt fst-conv list.size
  apply (intro nth-equalityI, force)
  subgoal for i apply (cases i, simp-all add: nth-append)
  by (metis Suc-lessI nth-Cons-Suc)
  done
qed (simp add: filt)
finally have Mapping.lookup (assign-by-sort Mapping.empty bnds) s =
  Some (snd (hd filt) - int (length filt) + 1, rev (map (λi. (fst (fst (filt ! i)),
snd (hd filt) - int i)) [0..<length filt]))
  unfolding filt
  by auto
} note ne-impl = this
have filt: filt = ?filt unfolding filt filt-def fi-def ..
show ?thesis
proof (cases filt = [])
  case False

```

```

thus ?thesis unfolding ne-impl[OF False] filt by auto
next
  case True
  thus ?thesis using filt unfolding impl filt-def[symmetric] by auto
qed
qed

```

definition *find-large-sorts* :: $((v \times s) \times \text{int}) \text{ list} \Rightarrow s \text{ set}$ **where**
find-large-sorts bnds = (let
 m = assign-by-sort Mapping.empty bnds;
 mf = Mapping.filter $(\lambda s (b,vs). b \geq 0)$ m
 in Mapping.keys mf)

Delete all variables of a sort where the upper bound is large enough to make all variables of this sort distinct. Afterwards also delete all non-occurring variables from the bounds-list.

definition *delete-large-sorts-single* :: $(v,s)\text{fidl-input} \Rightarrow (v,s)\text{fidl-input} \times \text{bool}$
where
delete-large-sorts-single = $(\lambda (bnds, diffs).$
 let lsorts = *find-large-sorts* bnds
 in if Set.is-empty lsorts then $((bnds, diffs), \text{False})$
 else let newdiffs = filter $(\lambda vs. \forall vw \in \text{set } vs. \text{snd } (fst \text{ } vw) \notin \text{lsorts})$ diffs;
 remvars = set (List.maps (List.maps $(\lambda (v,w). [v,w])$) newdiffs);
 newbnds = filter $(\lambda vb. fst \text{ } vb \in \text{remvars})$ bnds
 in $((\text{newbnds}, \text{newdiffs}), \text{True}))$

lemma *delete-large-sorts-single*: **assumes** *inp*: *fidl-input* (*input* :: $(v,s)\text{fidl-input}$)
and *del*: *delete-large-sorts-single* *input* = (*output*, *changed*)
shows *fidl-input* *output* \wedge
 (*fidl-solvable* *input* = *fidl-solvable* *output*) \wedge
 (*changed* \longrightarrow length (fst *input*) > length (fst *output*))

proof –

```

obtain bnds diffs where input: input = (bnds,diffs) by force
define lsorts where lsorts = find-large-sorts bnds
show ?thesis
proof (cases Set.is-empty lsorts)
  case True
  with del inp show ?thesis
  unfolding delete-large-sorts-single-def input split lsorts-def Let-def by auto
next
  case False
  define f where f vs =  $(\forall vw \in \text{set } vs. \text{snd } (fst \text{ } vw) \notin \text{lsorts})$  for vs ::  $((v \times s)$ 
   $\times v \times s)$  list
  define newdiffs where newdiffs = filter f diffs
  define remvars where remvars = set (List.maps (List.maps  $(\lambda (v, w). [v, w])$ )
  newdiffs)
  define newbnds where newbnds = filter  $(\lambda vb. fst \text{ } vb \in \text{remvars})$  bnds
  let ?out = (newbnds, newdiffs)

```

```

from del[unfolded delete-large-sorts-single-def input split] False
have outp: output = ?out
by (simp add: Let-def lsorts-def newdiffs-def remvars-def newbnds-def f-def[abs-def])
note inp = inp[unfolded input fidl-input-def split]
from inp have dist: distinct (map fst bnds) by blast
have nd-sub: set newdiffs  $\subseteq$  set diffs unfolding newdiffs-def by auto
have nb-sub: set newbnds  $\subseteq$  set bnds unfolding newbnds-def by auto
have fidl-out': fidl-input output
  unfolding fidl-input-def split outp
proof (intro conjI allI impI)
  show distinct (map fst newbnds) unfolding newbnds-def by (rule distinct-map-filter[OF dist])
  show  $\bigwedge v w b1 b2. (v, b1) \in \text{set newbnds} \implies (w, b2) \in \text{set newbnds} \implies \text{snd } v = \text{snd } w \implies b1 = b2$ 
    using nb-sub inp by blast
  show  $\bigwedge v b. (v, b) \in \text{set newbnds} \implies 0 \leq b$  using nb-sub inp by blast
  from nd-sub have csub: set (concat newdiffs)  $\subseteq$  set (concat diffs) by auto
  from csub show  $\bigwedge v w. (v, w) \in \text{set (concat newdiffs)} \implies \text{snd } v = \text{snd } w$ 
using inp by (meson in-mono)
  from csub show  $\bigwedge v w. (v, w) \in \text{set (concat newdiffs)} \implies v \neq w$  using inp
by auto
  fix v w u
  assume vw:  $(v, w) \in \text{set (concat newdiffs)}$  and u:  $u \in \{v, w\}$ 
  with csub inp have ubnds:  $u \in \text{fst ' set bnds}$  by (meson in-mono)
  from vw u have  $u \in \text{remvars}$  unfolding remvars-def
  by (cases v; cases w; cases u; force)
  with vw u ubnds show  $u \in \text{fst ' set newbnds}$  unfolding newbnds-def by force
qed
note fidl-out = fidl-out'[unfolded outp fidl-input-def split]
show ?thesis
proof (intro conjI impI fidl-out')
  let ?sat1 =  $\lambda \text{ bnds } \alpha. (\forall (v, b) \in \text{set bnds}. 0 \leq \alpha v \wedge \alpha v \leq b)$ 
  let ?sat2 =  $\lambda \text{ diffs } \alpha. (\forall c \in \text{set diffs}. \exists (v, w) \in \text{set } c. \alpha v \neq \alpha w)$ 
  let ?sat =  $\lambda \text{ bnds diffs } \alpha. ?sat1 \text{ bnds } \alpha \wedge ?sat2 \text{ diffs } \alpha$ 
  show fidl-solvable input = fidl-solvable output
proof
  assume fidl-solvable input
  then obtain  $\alpha$  where ?sat bnds diffs  $\alpha$  unfolding fidl-solvable-def input
by auto
  hence ?sat newbnds newdiffs  $\alpha$  using nd-sub nb-sub by auto
  thus fidl-solvable output unfolding fidl-solvable-def outp by auto
next
  assume fidl-solvable output
  from this[unfolded fidl-solvable-def outp] obtain  $\alpha$  where
    sat1: ?sat1 newbnds  $\alpha$  and sat2: ?sat2 newdiffs  $\alpha$  by auto
  define m where  $m = \text{assign-by-sort Mapping.empty bnds}$ 
  define filt where  $\text{filt } s = \text{filter } ((=) s \circ \text{snd} \circ \text{fst}) \text{ bnds}$  for s

```

```

define mf where mf = Mapping.filter ( $\lambda s (b,vs). b \geq 0$ ) m
{
  fix s i vs
  assume Mapping.lookup mf s = Some (i,vs)
  from this[unfolded mf-def] have look-m: Mapping.lookup m s = Some
(i,vs) and i0: i  $\geq 0$ 
  by (transfer, auto split: option.splits if-splits)+
  from look-m[unfolded m-def assign-by-sort-computation[OF filt-def]]
  have i: i = snd (hd (filt s)) - int (length (filt s)) + 1
  and vs: vs = rev (map ( $\lambda i. (fst (fst (filt s ! i)), snd (hd (filt s)) - int i)$ 
[ $0..<length (filt s)$ ]))
  and ne: filt s  $\neq []$  by (auto split: if-splits)
  from ne obtain x b s' fs where filt s =  $((x,s'),b) \# fs$ 
  by (cases filt s, auto)
  with arg-cong[OF this, of set, unfolded filt-def]
  have filt: filt s =  $((x,s),b) \# fs$  unfolding filt-def by (force simp: o-def)
  from arg-cong[OF filt[unfolded filt-def], of set]
  have in-bnds: ((x,s),b)  $\in$  set bnds by auto
  from filt have hd: hd (filt s) =  $((x,s),b)$  by auto
  note i = i[unfolded hd snd-conv]
  note vs = vs[unfolded hd snd-conv]
  from i0[unfolded i]
  have bnd: int (length (filt s))  $\leq b + 1$  by simp
  from arg-cong[OF vs, of  $\lambda s. snd ' set s$ ] bnd
  have bounded: snd ' set vs  $\subseteq \{0..b\}$  by auto
  have dist: distinct (map snd vs)
  unfolding vs rev-map[symmetric] distinct-rev
  unfolding map-map o-def snd-conv
  unfolding distinct-map by (auto simp: inj-on-def)
  have image: (fst o fst) ' set (filt s)  $\subseteq$  fst ' set vs
  unfolding vs set-map set-rev fst-conv image-comp o-def set-upt
  unfolding set-conv-nth by auto
  have  $\exists x b. ((x,s),b) \in set bnds \wedge snd ' set vs \subseteq \{0..b\}$  using bounded
in-bnds by auto
  note image dist this
} note mf-lookup = this

```

```

define  $\beta$  where  $\beta = (\lambda (v,s).$ 
  case Mapping.lookup mf s of None  $\Rightarrow$  if  $(v,s) \in fst ' set newbnds$  then  $\alpha$ 
(v,s) else 0 | Some (-,vs)  $\Rightarrow$  the (map-of vs v))
  have lsorts: lsorts = Mapping.keys mf unfolding mf-def m-def lsorts-def
find-large-sorts-def Let-def ..
  have ?sat1 bnds  $\beta$ 
proof (intro ballI, clarsimp)
  fix v s b
  assume usb: ((v,s),b)  $\in$  set bnds
  with inp have b0: b  $\geq 0$  by auto
  show  $0 \leq \beta (v, s) \wedge \beta (v, s) \leq b$ 

```

```

proof (cases s ∈ lsorts)
  case False
  hence βα: β (v,s) = (if (v,s) ∈ fst ‘ set newbnds then α (v,s) else 0)
  unfolding β-def split lsorts keys-dom-lookup by (cases Mapping.lookup
mf s, auto)
  show ?thesis
  proof (cases (v,s) ∈ fst ‘ set newbnds)
    case True
    with vsb have mem: ((v,s),b) ∈ set newbnds unfolding newbnds-def
by auto
    with βα have β (v,s) = α (v,s) by force
    from sat1[rule-format, OF mem] this show ?thesis by auto
  next
  case False
  with βα b0 show ?thesis by auto
  qed
next
  case True
  from this[unfolded lsorts keys-dom-lookup]
  obtain i vs where look-mf: Mapping.lookup mf s = Some (i,vs) by auto
  from vsb have v ∈ (fst ∘ fst) ‘ set (filt s) unfolding filt-def by force
  with mf-lookup[OF look-mf] obtain x b' where v: v ∈ fst ‘ set vs
    and x: ((x, s), b') ∈ set bnds and vs: snd ‘ set vs ⊆ {0..b'} by blast
  from x vsb inp have b' = b by auto
  note vs = vs[unfolded this]
  from v have map-of vs v ≠ None
    by (simp add: map-of-eq-None-iff)
  then obtain i where map: map-of vs v = Some i by blast
  from map-of-SomeD[OF this] vs have i: i ∈ {0..b} by auto
  have β: β (v,s) = the (map-of vs v) unfolding β-def look-mf split by
auto
  thus ?thesis unfolding map using i by auto
  qed
qed
moreover
have ?sat2 diffs β
proof
  fix c
  assume c: c ∈ set diffs
  show ∃(v, w) ∈ set c. β v ≠ β w
  proof (cases c ∈ set newdiffs)
    case True
    from this[unfolded newdiffs-def] have fc: f c by auto
    from True sat2 obtain v w where vw: (v,w) ∈ set c and diff: α v ≠ α
w by auto
    from fc[unfolded f-def] vw have sort: snd v ∉ lsorts by auto
    from fidl-out True vw have vw-bnds: {v,w} ⊆ fst ‘ set newbnds unfolding
set-concat
    by (smt (verit, ccfv-SIG) UnionI image-iff subset-eq)

```

```

    from fidl-out True vw have same-sort: snd v = snd w unfolding
set-concat by blast
  {
    fix u
    assume  $u \in \{v,w\}$ 
    with sort same-sort vw-bnds obtain  $x\ s$  where
       $u: u = (x,s)$  and  $s: s \notin \text{lsorts } (x,s) \in \text{fst ' set newbnds}$  by force
    have  $\beta\ u = \alpha\ u$  unfolding  $\beta\text{-def } u$  split using  $s$ 
      by (cases Mapping.lookup mf s, auto simp: lsorts keys-dom-lookup)
  }
  with vw diff show  $\exists (v, w) \in \text{set } c. \beta\ v \neq \beta\ w$ 
    by (intro bexI[of - (v,w)], auto)
next
case False
from this[unfolded newdiffs-def]  $c$  have  $\neg f\ c$  by auto
from this[unfolded f-def] obtain  $v\ w$  where
   $vw: (v,w) \in \text{set } c$  and sort: snd v \in lsorts by force
from vw inp c have  $\text{snd } v = \text{snd } w$  unfolding set-concat by blast
with sort obtain  $x\ y\ s$  where  $v: v = (x,s)$  and  $w: w = (y,s)$  and  $s: s \in \text{lsorts}$ 
  by (cases v; cases w; auto)
from s[unfolded lsorts keys-dom-lookup]
obtain  $i\ vs$  where look-mf: Mapping.lookup mf s = Some (i,vs) by auto
from vw inp c have  $v \neq w$  unfolding set-concat by blast
with  $v\ w$  have  $xy: x \neq y$  by auto
  have  $\beta v: \beta\ v = \text{the } (\text{map-of } vs\ x)$  unfolding  $\beta\text{-def } \text{look-mf}$  split  $v$  by
auto
  have  $\beta w: \beta\ w = \text{the } (\text{map-of } vs\ y)$  unfolding  $\beta\text{-def } \text{look-mf}$  split  $w$  by
auto
show  $\exists (v, w) \in \text{set } c. \beta\ v \neq \beta\ w$ 
proof (intro bexI[OF - vw], unfold split \beta v \beta w)
  from mf-lookup[OF look-mf]
  have  $\text{sub}: (\text{fst} \circ \text{fst})\ \text{' set } (\text{filt } s) \subseteq \text{fst ' set } vs$ 
    and dist: distinct (map snd vs) by auto
  from vw c have  $(v,w) \in \text{set } (\text{concat } \text{diffs})$  by auto
  with inp have  $vw: \{v,w\} \subseteq \text{fst ' set } \text{bnds}$  by blast
  {
    fix  $z$ 
    assume  $z \in \{x,y\}$ 
    hence  $(z,s) \in \text{fst ' set } \text{bnds}$  using  $vw$  unfolding  $v\ w$  by auto
    hence  $z \in (\text{fst} \circ \text{fst})\ \text{' set } (\text{filt } s)$  unfolding filt-def by force
    with sub have  $z \in \text{fst ' set } vs$  by auto
    hence  $\text{map-of } vs\ z \neq \text{None}$  by (simp add: map-of-eq-None-iff)
  }
  from this[of x] this[of y] obtain  $i\ j$  where
     $\text{map-of } vs\ x = \text{Some } i$   $\text{map-of } vs\ y = \text{Some } j$ 
    by auto
  hence  $\text{mem}: (x,i) \in \text{set } vs\ (y,j) \in \text{set } vs$  by (auto simp: map-of-SomeD)
  then obtain  $k\ l$  where  $*$ :  $k < \text{length } vs\ !\ k = (x,i)\ l < \text{length } vs\ vs$ 

```

```

! l = (y,j)
  unfolding set-conv-nth by auto
  from * xy have k ≠ l by auto
  with dist * have ij: i ≠ j unfolding distinct-conv-nth by force
  thus the (map-of vs x) ≠ the (map-of vs y) unfolding map by auto
  qed
  qed
  qed
  ultimately show fdl-solvable input unfolding input fdl-solvable-def by
auto
  qed
next
define m where m = assign-by-sort Mapping.empty bnds
define filt where filt s = filter ((=) s ∘ snd ∘ fst) bnds for s
define mf where mf = Mapping.filter (λ s (b,vs). b ≥ 0) m
from False obtain s where sl: s ∈ lsorts by auto
from this[unfolded lsorts-def find-large-sorts-def Let-def, folded m-def]
have s: s ∈ Mapping.keys m using keys-filter by fastforce
let ?f = ((=) s ∘ snd ∘ fst)
from s obtain b vs where Mapping.lookup m s = Some (b,vs)
  by (metis in-keysD surj-pair)
with assign-by-sort-computation[OF filt-def, folded m-def, of s] have set (filt
s) ≠ {}
  by (auto split: if-splits)
from this[unfolded filt-def] obtain e where fe: ?f e and e: e ∈ set bnds by
auto
from fe obtain v b where e-id: e = ((v,s),b) by (cases e, auto)
let ?v = (v,s)
from split-list[OF e] obtain bef aft where bnds: bnds = bef @ e # aft by
auto
have e: fst e ∉ remvars
proof
  assume fst e ∈ remvars
  from this[unfolded remvars-def e-id fst-conv] obtain w
    where (?v,w) ∈ set (concat newdiffs) ∨ (w,?v) ∈ set (concat newdiffs)
    by auto
  from this[unfolded newdiffs-def] obtain vs where f: f vs and vs: vs ∈ set
diffs
    and (?v,w) ∈ set vs ∨ (w,?v) ∈ set vs by auto
  from this(3) f[unfolded f-def] sl
  have (w,?v) ∈ set vs and snd: snd w ≠ s by auto
  from this(1) vs have (w,?v) ∈ set (concat diffs) by auto
  with inp have snd w = snd ?v by blast
  with snd show False by auto
qed
have length (fst output) = length (filter (λvb. fst vb ∈ remvars) (bef @ aft))
  unfolding outp fst-conv newbnds-def bnds using e by auto
also have ... ≤ length (bef @ aft) by (rule length-filter-le)
also have ... < length (fst input) unfolding input bnds by auto

```

```

    finally show length (fst output) < length (fst input) .
  qed
  qed
  qed

```

```

partial-function (tailrec) delete-large-sorts :: ('v,'s)fidl-input ⇒ ('v,'s)fidl-input
where
  [code]: delete-large-sorts inp = (case delete-large-sorts-single inp of
    (out,changed) ⇒ if changed then delete-large-sorts out else out)

```

```

lemma delete-large-sorts: assumes fidl-input inp
  and del: delete-large-sorts inp = out
shows fidl-input out ∧ fidl-solvable inp = fidl-solvable out
  using assms
proof (induct inp rule: wf-induct[OF wf-measure, of length o fst])
  case (1 inp)
  obtain mid ch where single: delete-large-sorts-single inp = (mid,ch) (is ?e = -)
by (cases ?e, auto)
  from delete-large-sorts-single[OF 1(2) this]
  have *: fidl-input mid
    fidl-solvable inp = fidl-solvable mid
    ch ⇒ length (fst mid) < length (fst inp)
  by auto
  note out = 1(3)[unfolded delete-large-sorts.simps[of inp, unfolded single split]]
  show ?case
  proof (cases ch)
  case False
  with out * show ?thesis by auto
  next
  case True
  with out have out: delete-large-sorts mid = out by auto
  from *(3)[OF True] have (mid, inp) ∈ measure (length o fst) by auto
  from 1(1)[rule-format, OF this *(1) out] * show ?thesis by auto
  qed
  qed

```

```

definition fidl-pre-processor where
  fidl-pre-processor solver input = (case delete-trivial-sorts input
    of None ⇒ False
    | Some mid ⇒ let (bnds',diffs') = delete-large-sorts mid
      in if bnds' = [] ∧ diffs' = [] then True else solver (bnds', diffs'))

```

```

lemma fidl-pre-processor: assumes finite-idl-solver solver
  shows finite-idl-solver (fidl-pre-processor solver)
  unfolding finite-idl-solver-def
proof (intro allI impI)
  fix input :: ('a,'b)fidl-input
  assume bd: fidl-input input
  note triv = delete-trivial-sorts[OF bd]

```

```

note  $res = \text{fidl-pre-processor-def}[of\ solver\ input]$ 
show  $\text{fidl-pre-processor}\ solver\ input = \text{fidl-solvable}\ input$ 
proof (cases delete-trivial-sorts input)
  case None
  with  $triv\ res$  show  $?thesis$  by auto
next
  case (Some mid)
  from  $triv[OF\ Some, of\ mid]$ 
  have  $mid: \text{fidl-input}\ mid$  and  $bd: \text{fidl-solvable}\ input = \text{fidl-solvable}\ mid$  by auto
  note  $res = res[unfolded\ Some\ option.simps]$ 
  obtain  $ob\ od$  where  $large: \text{delete-large-sorts}\ mid = (ob, od)$  by force
  note  $res = res[unfolded\ large\ Let-def\ split]$ 
  from  $\text{delete-large-sorts}[OF\ mid\ large]$ 
  have  $out: \text{fidl-input}\ (ob, od)$  and  $mid: \text{fidl-solvable}\ mid = \text{fidl-solvable}\ (ob, od)$ 
by auto
  show  $?thesis$ 
  proof (cases  $ob = [] \wedge od = []$ )
    case True
    hence  $\text{fidl-solvable}\ (ob, od)$  unfolding  $\text{fidl-solvable-def}$  by auto
    with  $True\ res$  show  $?thesis$  unfolding  $bd\ mid$  by auto
  next
  case False
  from  $assms[unfolded\ finite-idl-solver-def, rule-format, OF\ out]$ 
  show  $?thesis$  using  $False$  unfolding  $bd\ mid\ res$  by auto
  qed
qed
qed

```

```

datatype  $'v\ \text{fidl-constraint} = \text{Var-Int}\ (\text{fidlc-vi}: 'v \times \text{int}) \mid \text{Var-Var}\ (\text{fidlc-vv}: 'v \times 'v)$ 

```

```

fun  $is\ \text{fidlc-vi}$  where
   $is\ \text{fidlc-vi}\ (\text{Var-Int}\ -) = True$ 
|  $is\ \text{fidlc-vi}\ - = False$ 

```

```

fun  $\text{fidlc-sat} :: ('v \Rightarrow \text{int}) \Rightarrow 'v\ \text{fidl-constraint} \Rightarrow \text{bool}$  where
   $\text{fidlc-sat}\ \alpha\ (\text{Var-Int}\ (v, i)) = (\alpha\ v \neq i)$ 
|  $\text{fidlc-sat}\ \alpha\ (\text{Var-Var}\ (v, w)) = (\alpha\ v \neq \alpha\ w)$ 

```

```

fun  $\text{fidlc-vars} :: 'v\ \text{fidl-constraint} \Rightarrow 'v\ \text{list}$  where
   $\text{fidlc-vars}\ (\text{Var-Int}\ (v, i)) = [v]$ 
|  $\text{fidlc-vars}\ (\text{Var-Var}\ (v, w)) = [v, w]$ 

```

```

lemma  $\text{fidlc-vi}[simp]: is\ \text{fidlc-vi}\ vi \Longrightarrow \text{Var-Int}\ (\text{fidlc-vi}\ vi) = vi$ 
by (cases  $vi$ , auto)

```

```

lemma  $\text{fidlc-vv}[simp]: \neg is\ \text{fidlc-vi}\ vv \Longrightarrow \text{Var-Var}\ (\text{fidlc-vv}\ vv) = vv$ 
by (cases  $vv$ , auto)

```

```

datatype 'v fidl-constraints = IDL-CS
  ('v × int)list
  ('v × 'v)list
  'v fidl-constraint list list

fun fidl-flat-cs :: 'v fidl-constraints ⇒ 'v fidl-constraint set set where
  fidl-flat-cs (IDL-CS vis vws cs) = set (map set (map (singleton o Var-Int) vis @
  map (singleton o Var-Var) vws @ cs))

fun fidl-cs-restructure :: 'v fidl-constraints ⇒ 'v fidl-constraints option where
  fidl-cs-restructure (IDL-CS vis vws cs) = (if [] ∈ set cs then None else Some (
  case partition is-singleton-list cs of
  (ss, other) ⇒ (case partition is-fidlc-vi (map hd ss)
  of (xis,xys) ⇒ (IDL-CS (map fidlc-vi xis @ vis) (map fidlc-vv xys @ vws)
  other))))

definition fidl-flat-vs :: 'v fidl-constraints ⇒ 'v set where
  fidl-flat-vs C = (∪ c ∈ fidl-flat-cs C. (∪ a ∈ c. set (fidlc-vars a)))

definition fidl-constraints-sat :: 'v fidl-constraints ⇒ ('v ⇒ int) ⇒ bool where
  fidl-constraints-sat cs α = (∀ disj ∈ fidl-flat-cs cs. Bex disj (fidlc-sat α))

lemma fidl-cs-restructure: assumes fidl-cs-restructure cs = cso
  shows cso = None ⇒ ¬ (Ex (fidl-constraints-sat cs))
  cso = Some cs' ⇒ fidl-flat-cs cs = fidl-flat-cs cs'
  cso = Some cs' ⇒ fidl-constraints-sat cs = fidl-constraints-sat cs'
  cso = Some cs' ⇒ fidl-flat-vs cs = fidl-flat-vs cs'

proof –
  obtain vis vws css where cs: cs = IDL-CS vis vws css by (cases cs, auto)
  {
  assume cso = None
  with assms[unfolded cs, simplified] have [] ∈ set css by (auto split: if-splits)
  thus ¬ (Ex (fidl-constraints-sat cs)) unfolding cs by (auto simp: fidl-constraints-sat-def)
  }
  obtain ss other where p1: partition is-singleton-list css = (ss,other) by force
  obtain xis xys where p2: partition is-fidlc-vi (map hd ss) = (xis,xys) by force
  have css: set css = set ss ∪ set other using p1 by auto
  have id: map set ss = map (set o singleton) (map hd ss) unfolding map-map
  proof (rule map-cong[OF refl])
  fix s
  assume s ∈ set ss with p1 have is-singleton-list s by auto
  thus set s = (set o singleton o hd) s using is-singleton-list[of s] by auto
  qed
  have ss: set ' set ss = (set o singleton) ' set (map hd ss)
  unfolding image-set unfolding id by auto
  have mss: set (map hd ss) = set xis ∪ set xys using p2 by auto
  have vi: (singleton o Var-Int) ' fidlc-vi ' set xis = singleton ' set xis
  unfolding image-comp o-def using p2 by auto

```

```

have vv: (singleton ◦ Var-Var) ‘ fidlc-vv ‘ set xys = singleton ‘ set xys
  unfolding image-comp o-def using p2 by auto
assume cso = Some cs'
with assms[unfolded cs fidl-cs-restructure.simps p1 split p2]
have cs': cs' = IDL-CS (map fidlc-vi xis @ vis) (map fidlc-vv xys @ vws) other
by (auto split: if-splits)
show fidl-flat-cs cs = fidl-flat-cs cs'
  unfolding cs cs' fidl-flat-cs.simps map-append set-map set-append css image-Un

  unfolding ss mss vi vv by auto
thus fidl-constraints-sat cs = fidl-constraints-sat cs' fidl-flat-vs cs = fidl-flat-vs
cs'
  unfolding fidl-constraints-sat-def fidl-flat-vs-def by auto
qed

```

```

datatype 'v fidl-solver-state =
  IDL-State ('v,int list)mapping 'v fidl-constraints

```

```

fun fidl-state-sat :: 'v fidl-solver-state ⇒ ('v ⇒ int) ⇒ bool where
  fidl-state-sat (IDL-State bnds cs) α = (
    (∀ v bnd. Mapping.lookup bnds v = Some bnd → α v ∈ set bnd)
    ∧ fidl-constraints-sat cs α)

```

```

fun fidl-state :: 'v fidl-solver-state ⇒ 'v set ⇒ bool where
  fidl-state (IDL-State bnds cs) V = (
    Ball (Mapping.entries bnds) (λ (v,ints). distinct ints ∧ (v ∈ V ∨ ints ≠ []))
    ∧ fidl-flat-vs cs ⊆ Mapping.keys bnds
    ∧ finite (Mapping.keys bnds))

```

```

fun fidl-vars :: 'v fidl-solver-state ⇒ 'v set where
  fidl-vars (IDL-State bnds cs) = Mapping.keys bnds

```

```

fun fidl-size :: 'v fidl-solver-state ⇒ nat where
  fidl-size (IDL-State bnds cs) = Mapping.size bnds

```

```

fun fidl-restructure :: 'v fidl-solver-state ⇒ 'v fidl-solver-state option where
  fidl-restructure (IDL-State bnds cs) = map-option (IDL-State bnds) (fidl-cs-restructure
cs)

```

```

fun fidl-delete-vi :: 'v fidl-solver-state ⇒ 'v fidl-solver-state × 'v list where
  fidl-delete-vi (IDL-State bnds (IDL-CS ((v,i) # vis) vws cs)) = (
    map-prod id (Cons v) (fidl-delete-vi (IDL-State (Mapping.map-entry v (remove1
i) bnds) (IDL-CS vis vws cs))))
| fidl-delete-vi (IDL-State bnds (IDL-CS [] vws cs)) = (
  (IDL-State bnds (IDL-CS [] vws cs)), [])

```

lemma *mapping-size-map-entry*[simp]: *Mapping.size (Mapping.map-entry x f m) = Mapping.size m*
unfolding *Mapping.size-def keys-map-entry by auto*

lemma *fidl-delete-vi*: **assumes** *fidl-delete-vi C = (C',xs) fidl-state C V*
shows *fidl-state C' (V ∪ set xs) fidl-state-sat C' = fidl-state-sat C*
proof –
have *fidl-state C' (V ∪ set xs) ∧ fidl-state-sat C' α = fidl-state-sat C α* **for** *α*
using *assms*
proof (*induct C arbitrary: C' xs V rule: fidl-delete-vi.induct*)
case (*1 bnds v i vis vws cs C xs' V*)
define *bnds'* **where** *bnds' = Mapping.map-entry v (remove1 i) bnds*
from *1(2)[unfolded fidl-delete-vi.simps bnds'-def[symmetric]]*
obtain *xs* **where** *rec: fidl-delete-vi (IDL-State bnds' (IDL-CS vis vws cs)) = (C,xs) (is ?e = -)*
and *xs': xs' = v # xs* **by** (*cases ?e, auto*)
have *sub: fidl-flat-vs (IDL-CS vis vws cs) ⊆ fidl-flat-vs (IDL-CS ((v, i) # vis) vws cs)*
unfolding *fidl-flat-vs-def by auto*
have *state: fidl-state (IDL-State bnds' (IDL-CS vis vws cs)) (insert v V)*
unfolding *bnds'-def fidl-state.simps keys-map-entry*
unfolding *bnds'-def[symmetric]*
proof (*intro conjI ballI*)
show *fidl-flat-vs (IDL-CS vis vws cs) ⊆ Mapping.keys bnds*
using *1(3) sub by auto*
fix *xb*
assume *xb ∈ Mapping.entries bnds'*
then obtain *x b* **where** *xb: xb = (x,b)* **and** *look: Mapping.lookup bnds' x = Some b*
by (*cases xb, auto dest: in-entriesD*)
note *look = look[unfolded lookup-map-entry' bnds'-def]*
then obtain *b'* **where** *look: Mapping.lookup bnds x = Some b'* **and** *sub: b = b' ∨ (x = v ∧ b = remove1 i b')*
using *set-remove1-subset[of i]*
by (*cases Mapping.lookup bnds x; cases v = x, auto*)
from *look* **have** *(x,b') ∈ Mapping.entries bnds* **by** (*rule in-entriesI*)
with *1(3)[simplified]* **have** *distinct b'* **and** *V: (x ∈ V ∨ b' ≠ [])* **by** *auto*
with *sub* **have** *distinct b* **by** *auto*
with *V sub* **show** *case xb of (x, b) ⇒ distinct b ∧ (x ∈ insert v V ∨ b ≠ [])*
unfolding *xb by auto*
qed (*insert 1(3), auto*)

note *IH = 1(1)[OF rec[unfolded bnds'-def], folded bnds'-def, OF state]*

have *fidl-state-sat (IDL-State bnds (IDL-CS ((v, i) # vis) vws cs)) α*
= ((α v ≠ i ∧ (∀ v ints. Mapping.lookup bnds v = Some ints → α v ∈ set ints))
∧ fidl-constraints-sat (IDL-CS vis vws cs) α)
unfolding *fidl-state-sat.simps*

```

    by (auto simp add: fdl-constraints-sat-def singleton-def)
  also have ( $\alpha v \neq i \wedge (\forall v \text{ ints. Mapping.lookup bnds } v = \text{Some ints} \longrightarrow \alpha v \in \text{set ints})$ )
    = ( $\forall v \text{ ints. Mapping.lookup bnds}' v = \text{Some ints} \longrightarrow \alpha v \in \text{set ints}$ ) (is ?l
  = ?r)
  proof
    assume ?l
    show ?r
    proof (intro allI conjI impI)
      fix w ints
      assume Mapping.lookup bnds' w = Some ints
      from this[unfolded lookup-map-entry' bnds'-def]
      show  $\alpha w \in \text{set ints}$ 
        using  $\langle ?l \rangle$ [THEN conjunct2, rule-format, of w]  $\langle ?l \rangle$ [THEN conjunct1]
        by (cases Mapping.lookup bnds w; cases v = w, auto)
    qed
  next
    assume ?r
    show ?l
    proof (intro conjI allI impI)
      fix w ints
      assume Mapping.lookup bnds w = Some ints
      with  $\langle ?r \rangle$ [rule-format, of w, unfolded bnds'-def lookup-map-entry']
      show  $\alpha w \in \text{set ints}$  using set-remove1-subset[of i]
        by (cases v = w, auto)
    next
      from 1(3) have v:  $v \in \text{Mapping.keys bnds}$  by (auto simp: fdl-flat-vs-def
    singleton-def)
      then obtain ints where look: Mapping.lookup bnds v = Some ints by
    (meson in-keysD)
      hence (v,ints)  $\in \text{Mapping.entries bnds}$  by (rule in-entriesI)
      with 1(3) have distinct ints by auto
      from  $\langle ?r \rangle$ [rule-format, of v, unfolded bnds'-def Mapping.lookup-map-entry,
    unfolded look]
      set-remove1-eq[OF this]
      show  $\alpha v \neq i$  by simp
    qed
  qed
  also have (...  $\wedge$  fdl-constraints-sat (IDL-CS vis vws cs)  $\alpha$ )
    = fdl-state-sat (IDL-State bnds' (IDL-CS vis vws cs))  $\alpha$ 
    by (auto simp add: fdl-constraints-sat-def singleton-def)
  finally have equiv: fdl-state-sat (IDL-State bnds (IDL-CS ((v, i) # vis) vws
  cs))  $\alpha$  =
    fdl-state-sat (IDL-State bnds' (IDL-CS vis vws cs))  $\alpha$  .
  show ?case unfolding equiv using IH xs' unfolding bnds'-def fdl-size.simps
  by auto
  qed auto
  thus fdl-state C' (V  $\cup$  set xs) fdl-state-sat C' = fdl-state-sat C by auto
  qed

```

lemma *fidl-delete-vi-size*: $\text{fidl-delete-vi } C = (C', vs) \implies \text{fidl-size } C' \leq \text{fidl-size } C$
proof –
have $\text{fidl-size } (\text{fst } (\text{fidl-delete-vi } C)) = \text{fidl-size } C$
by (*induct C rule: fidl-delete-vi.induct, auto*)
thus $\text{fidl-delete-vi } C = (C', vs) \implies \text{fidl-size } C' \leq \text{fidl-size } C$ **by** *auto*
qed

lemma *fidl-restructure*: $\text{fidl-restructure } s = \text{None} \implies \neg \text{Ex } (\text{fidl-state-sat } s)$
 $\text{fidl-restructure } s = \text{Some } s' \implies \text{fidl-state-sat } s' = \text{fidl-state-sat } s$
 $\text{fidl-restructure } s = \text{Some } s' \implies \text{fidl-state } s' = \text{fidl-state } s$
 $\text{fidl-restructure } s = \text{Some } s' \implies \text{fidl-size } s' = \text{fidl-size } s$
proof –
obtain *bnds cs where s: s = IDL-State bnds cs by (cases s, auto)*
note $re = \text{fidl-cs-restructure}[OF \text{ refl, of cs}]$
note $def = \text{fidl-restructure.simps}[of \text{ bnds cs, folded s}]$
 {
assume $\text{fidl-restructure } s = \text{None}$
with $re(1) \text{ def show } \neg \text{Ex } (\text{fidl-state-sat } s)$ **unfolding** s **by** *auto*
 }
assume $\text{fidl-restructure } s = \text{Some } s'$
with def **obtain** cs' **where** $cs: \text{fidl-cs-restructure } cs = \text{Some } cs'$ **and** $s': s' = \text{IDL-State bnds } cs'$
by (*cases fidl-cs-restructure cs, auto*)
note $re = re(2-)[OF \text{ cs}]$
from re **show** $\text{fidl-state-sat } s' = \text{fidl-state-sat } s$ **unfolding** s' **by** *auto*
from re **show** $\text{fidl-state } s' = \text{fidl-state } s$ **unfolding** s' **by** *auto*
show $\text{fidl-size } s' = \text{fidl-size } s$ **unfolding** s' **by** *simp*
qed

lemma *all-entries-eq-all-lookups*:
 $(\forall (x, i) \in \text{Mapping.entries } m. P \ x \ i) = (\forall x \ i. \text{Mapping.lookup } m \ x = \text{Some } i \longrightarrow P \ x \ i)$
by (*metis case-prodI2 case-prod-conv in-entriesD in-entriesI*)

fun *inst-vv where inst-vv v i [] = Some ([], [])*
 | $\text{inst-vv } v \ i \ ((x, y) \# \ xs) = (\text{if } x = v \ \text{then } (\text{if } y = v \ \text{then } \text{None}$
 $\text{else } \text{map-option } (\text{map-prod } (\text{Cons } (y, i)) \ \text{id}) \ (\text{inst-vv } v \ i \ \xs))$
 $\text{else } \text{if } y = v \ \text{then } \text{map-option } (\text{map-prod } (\text{Cons } (x, i)) \ \text{id}) \ (\text{inst-vv } v \ i \ \xs)$
 $\text{else } \text{map-option } (\text{map-prod } \ \text{id} \ (\text{Cons } (x, y))) \ (\text{inst-vv } v \ i \ \xs))$

lemma *inst-vv: assumes $\alpha \ v = i$*
shows $\text{inst-vv } v \ i \ vvs = \text{None} \implies \neg \text{Ball } (\text{set } (\text{map } \text{Var-Var } vvs)) \ (\text{fidlc-sat } \alpha)$
 $\text{inst-vv } v \ i \ vvs = \text{Some } (vis, nvvs) \implies \text{Ball } (\text{set } (\text{map } \text{Var-Var } vvs)) \ (\text{fidlc-sat } \alpha)$
 $= (\text{Ball } (\text{set } (\text{map } \text{Var-Int } vis)) \ (\text{fidlc-sat } \alpha) \wedge \text{Ball } (\text{set } (\text{map } \text{Var-Var } nvvs)) \ (\text{fidlc-sat } \alpha))$

$inst\text{-}vv\ v\ i\ vvs = Some\ (vis,\ nvvs) \implies set\ (concat\ (map\ (fidlc\text{-}vars\ o\ Var\text{-}Int\ vis)) \cup set\ (concat\ (map\ (fidlc\text{-}vars\ o\ Var\text{-}Var\ nvvs)))$
 $\subseteq set\ (concat\ (map\ (fidlc\text{-}vars\ o\ Var\text{-}Var\ vvs)) - \{v\}$

proof (*atomize (full), induct vvs arbitrary: vis nvvs*)
case (*Cons xy xs fvis fvvs*)
obtain $x\ y$ **where** $xy: xy = (x,y)$ **by** *force*
show *?case*
proof (*cases x = v \wedge y = v*)
case *True*
thus *?thesis* **by** (*auto simp: xy*)
next
case *False*
show *?thesis*
proof (*cases inst-vv v i xs*)
case *None*
from *Cons(1)[THEN conjunct1, rule-format, OF this]*
show *?thesis using None xy by auto*
next
case (*Some pair*)
then obtain $nvis\ nvvs$ **where** $inst: inst\text{-}vv\ v\ i\ xs = Some\ (nvis,\ nvvs)$ **by**
(*cases pair, auto*)
have $simp: inst\text{-}vv\ v\ i\ (xy\ \# \ xs) = (if\ x = v\ then\ if\ y = v\ then\ None\ else\ Some\ ((y,\ i)\ \# \ nvis,\ nvvs)$
 $else\ if\ y = v\ then\ Some\ ((x,\ i)\ \# \ nvis,\ nvvs)\ else\ Some\ (nvis,\ (x,\ y)\ \# \ nvvs)$
unfolding $xy\ inst\text{-}vv.simps\ inst$ **by** *simp*
note $IH = Cons(1)[THEN\ conjunct2]$
note $IH = IH[THEN\ conjunct1,\ rule\text{-}format,\ OF\ inst]\ IH[THEN\ conjunct2,\ rule\text{-}format,\ OF\ inst]$
from *False* **have** $nN: inst\text{-}vv\ v\ i\ (xy\ \# \ xs) \neq None$ **unfolding** *simp* **by** *auto*
show *?thesis*
proof (*intro conjI impI*)
have $(\forall a \in set\ (map\ Var\text{-}Var\ (xy\ \# \ xs)).\ fidlc\text{-}sat\ \alpha\ a)$
 $= (\alpha\ x \neq \alpha\ y \wedge ((\forall a \in set\ (map\ Var\text{-}Var\ xs).\ fidlc\text{-}sat\ \alpha\ a)))$
unfolding xy **by** *auto*
note $IH\text{-}sat = this[unfolded\ IH(1)]$
assume $inst\text{-}vv\ v\ i\ (xy\ \# \ xs) = Some\ (fvis,\ fvvs)$
from $this[unfolded\ simp]$ *False*
have $eq: (fvis,\ fvvs) = (if\ x = v\ then\ ((y,\ i)\ \# \ nvis,\ nvvs)$
 $else\ if\ y = v\ then\ ((x,\ i)\ \# \ nvis,\ nvvs)\ else\ (nvis,\ (x,\ y)\ \# \ nvvs))$ **by** *auto*
show $(\forall a \in set\ (map\ Var\text{-}Var\ (xy\ \# \ xs)).\ fidlc\text{-}sat\ \alpha\ a) =$
 $((\forall a \in set\ (map\ Var\text{-}Int\ fvis).\ fidlc\text{-}sat\ \alpha\ a) \wedge (\forall a \in set\ (map\ Var\text{-}Var\ fvvs).\ fidlc\text{-}sat\ \alpha\ a))$
unfolding $IH\text{-}sat$ **using** $eq\ False\ assms$
by (*cases x = v; cases y = v; auto*)
show $set\ (concat\ (map\ (fidlc\text{-}vars\ o\ Var\text{-}Int)\ fvis)) \cup set\ (concat\ (map\ (fidlc\text{-}vars\ o\ Var\text{-}Var)\ fvvs))$
 $\subseteq set\ (concat\ (map\ (fidlc\text{-}vars\ o\ Var\text{-}Var)\ (xy\ \# \ xs))) - \{v\}$
using $eq\ IH(2)\ False\ xy$

```

      by (cases x = v; cases y = v; auto)
    qed (insert nN, auto)
  qed
  qed
  qed auto

fun inst-fidlc :: 'v ⇒ int ⇒ 'v fidl-constraint list ⇒ 'v fidl-constraint list option
where
  inst-fidlc v i [] = Some []
| inst-fidlc v i (Var-Var (x,y) # xs) = (if x = v then (if y = v then inst-fidlc v i xs else
  map-option (Cons (Var-Int (y,i))) (inst-fidlc v i xs))
  else if y = v then map-option (Cons (Var-Int (x,i))) (inst-fidlc v i xs)
  else map-option (Cons (Var-Var (x,y))) (inst-fidlc v i xs))
| inst-fidlc v i (Var-Int (x,j) # xs) = (if v = x then (if i = j then inst-fidlc v i xs
  else None)
  else map-option (Cons (Var-Int (x,j))) (inst-fidlc v i xs))

fun inst-fidlc-list :: 'v ⇒ int ⇒ 'v fidl-constraint list list ⇒ 'v fidl-constraint list
list where
  inst-fidlc-list v i [] = []
| inst-fidlc-list v i (vs # vvs) = (case inst-fidlc v i vs of
  None ⇒ inst-fidlc-list v i vvs
  | Some vs' ⇒ vs' # inst-fidlc-list v i vvs)

lemma inst-fidlc: assumes α v = i
shows inst-fidlc v i cs = None ⇒ Bex (set cs) (fidlc-sat α)
  inst-fidlc v i cs = Some cs' ⇒ Bex (set cs') (fidlc-sat α) = Bex (set cs)
(fidlc-sat α)
  inst-fidlc v i cs = Some cs' ⇒ set (concat (map fidlc-vars cs')) ⊆ set (concat
(map fidlc-vars cs)) - {v}
proof (atomize(full), induct cs arbitrary: cs')
case (Cons a xs)
note IH-None = Cons[THEN conjunct1, rule-format]
note IH-Some-sat = Cons[THEN conjunct2, THEN conjunct1, rule-format]
note IH-Some-vs = Cons[THEN conjunct2, THEN conjunct2, rule-format]
have bex: Bex (set (a # xs)) (fidlc-sat α) = (fidlc-sat α a ∨ Bex (set xs) (fidlc-sat
α)) by auto
show ?case
proof (cases a)
case (Var-Int xj)
then obtain x j where a: a = Var-Int (x,j) by (cases xj, auto)
show ?thesis
proof (cases inst-fidlc v i xs)
case None
with IH-None[OF this] assms
show ?thesis unfolding a bex by auto
next
case (Some ys)

```

```

    show ?thesis unfolding bex IH-Some-sat[OF Some, symmetric]
      using IH-Some-vs[OF Some]
      unfolding inst-fidlc.simps a Some using assms
      by (cases x = v; cases i = j; force)
  qed
next
case (Var-Var xy)
then obtain x y where a: a = Var-Var (x,y) by (cases xy, auto)
show ?thesis
proof (cases inst-fidlc v i xs)
  case None
  with IH-None[OF this] assms
  show ?thesis unfolding a bex by auto
next
case (Some ys)
show ?thesis unfolding bex IH-Some-sat[OF Some, symmetric]
  using IH-Some-vs[OF Some]
  unfolding inst-fidlc.simps a Some using assms
  by (cases x = v; cases y = v; force)
qed
qed
qed auto

lemma inst-fidlc-list: assumes  $\alpha v = i$ 
  shows  $\text{Ball} (\text{set} (\text{inst-fidlc-list } v \ i \ ccs)) (\lambda \text{ cs. } \text{Bex} (\text{set } \text{cs}) (\text{fidlc-sat } \alpha))$ 
  =  $\text{Ball} (\text{set } \text{ccs}) (\lambda \text{ cs. } \text{Bex} (\text{set } \text{cs}) (\text{fidlc-sat } \alpha))$ 
  set (concat (concat (map (map fidlc-vars) (inst-fidlc-list v i ccs))))  $\subseteq$  set (concat
(concat (map (map fidlc-vars) ccs))) - {v}
proof (atomize(full), induct ccs)
  case (Cons cs ccs)
  note step = inst-fidlc[of  $\alpha$ , OF assms, of cs]
  show ?case
  proof (cases inst-fidlc v i cs)
    case None
    with step(1)[OF this] Cons show ?thesis by auto
  next
  case (Some cs')
  show ?thesis
  proof (intro conjI)
    show  $(\forall \text{ cs} \in \text{set} (\text{inst-fidlc-list } v \ i \ (\text{cs} \# \text{ccs})). \exists a \in \text{set } \text{cs. } \text{fidlc-sat } \alpha \ a) =$ 
       $(\forall \text{ cs} \in \text{set} (\text{cs} \# \text{ccs}). \exists a \in \text{set } \text{cs. } \text{fidlc-sat } \alpha \ a)$ 
      using step(2)[OF Some] Cons[THEN conjunct1] Some by auto
    show set (concat (concat (map (map fidlc-vars) (inst-fidlc-list v i (cs #
ccs)))))
       $\subseteq$  set (concat (concat (map (map fidlc-vars) (cs # ccs)))) - {v}
      using step(3)[OF Some] Cons[THEN conjunct2] unfolding inst-fidlc-list.simps
Some option.simps
      list.simps concat.simps
      unfolding set-concat set-map set-append by fastforce

```

```

qed
qed
qed auto

```

```

fun instantiate-var :: 'v ⇒ int ⇒ 'v fidl-solver-state ⇒ 'v fidl-solver-state option
where
  instantiate-var v i (IDL-State bnds (IDL-CS vis vws cs)) = (
    case partition (((=) v) o fst) vis of
      (cvis,nvis1) ⇒ if i ∈ set (map snd cvis) then None
    else (case inst-vv v i vws of None ⇒ None
          | Some (nvis2, nvws) ⇒ let
              ncs = inst-fidlc-list v i cs
            in if [] ∈ set ncs then None
              else Some (IDL-State (Mapping.delete v bnds) (IDL-CS (nvis1 @ nvis2)
                nvws ncs))))

```

```

lemma lookup-delete-upd: Mapping.lookup (Mapping.delete x m) = (Mapping.lookup
m) (x := None)
apply (intro ext)
subgoal for y by (cases x = y, auto)
done

```

```

lemma instantiate-var: assumes α v = i
and Mapping.lookup bnds v = Some ints
and i ∈ set ints
and instantiate-var v i (IDL-State bnds css) = so
and fidl-state (IDL-State bnds css) (insert v V)
shows so = None ⇒ ¬ fidl-state-sat (IDL-State bnds css) α
  so = Some s ⇒ fidl-state s V ∧ fidl-state-sat (IDL-State bnds css) α = fidl-state-sat
s α
  ∧ fidl-vars s = fidl-vars (IDL-State bnds css) - {v}
  ∧ fidl-size s < fidl-size (IDL-State bnds css)
proof (atomize(full), goal-cases)
case 1
obtain vis vws cs where css: css = IDL-CS vis vws cs by (cases css, auto)
obtain cvis nvis1 where p1: partition (((=) v) o fst) vis = (cvis,nvis1) by force
let ?res = instantiate-var v i (IDL-State bnds css)
note res = instantiate-var.simps[of v i bnds vis vws cs, folded css, unfolded p1
split]
show ?case
proof (cases i ∈ set (map snd cvis))
case True
with res have res: ?res = None by auto
from True p1 have (v,i) ∈ set vis by auto
from split-list[OF this] assms(1)
have ¬ fidl-state-sat (IDL-State bnds css) α unfolding css
by (auto simp: fidl-constraints-sat-def singleton-def)
with res show ?thesis using assms by auto

```

```

next
  case cvis: False
  hence  $(i \in \text{set } (\text{map } \text{snd } \text{cvis})) = \text{False}$  by auto
  note res = res[unfolded this if-False]
  note inst-vv = inst-vv[of  $\alpha$ , OF assms(1)]
  show ?thesis
  proof (cases inst-vv v i vws)
    case None
    from inst-vv(1)[OF this] res[unfolded this] assms(4)
    show ?thesis by (auto simp: css fidl-constraints-sat-def singleton-def)
  next
  case (Some pairs)
  then obtain nvis2 nvws where ivv: inst-vv v i vws = Some (nvis2, nvws)
by force
  define ncs where ncs = inst-fidlc-list v i cs
  note res = res[unfolded ivv option.simps split Let-def, folded ncs-def]
  note inst-vv = inst-vv(2-3)[OF ivv]
  note inst-cs = inst-fidlc-list[of  $\alpha$  v i cs, folded ncs-def, OF assms(1)]
  show ?thesis
  proof (cases [] ∈ set ncs)
    case True
    with inst-cs have unsat:  $\neg (\forall cs \in \text{set } cs. \text{Bex } (\text{set } cs) (\text{fidlc-sat } \alpha))$  by auto
    from True res assms(4) have res: so = None by auto
    from unsat have  $\neg \text{fidl-state-sat } (\text{IDL-State } \text{bnds } \text{css}) \alpha$ 
      unfolding css by (auto simp: fidl-constraints-sat-def)
    with res show ?thesis by auto
  next
  case False
  with res assms(4)
  have so: so = Some (IDL-State (Mapping.delete v bnds) (IDL-CS (nvis1 @
nvis2) nvws ncs))
    (is - = Some ?s)
    by auto
  show ?thesis
  proof (intro conjI impI)
    show so = None  $\implies \neg \text{fidl-state-sat } (\text{IDL-State } \text{bnds } \text{css}) \alpha$  using so by
auto

    assume so = Some s
    with so have s: s = ?s by auto
    show fidl-state-sat (IDL-State bnds css) α = fidl-state-sat s α unfolding
s css
      fidl-state-sat.simps
    proof (intro arg-cong2[of - - - (∧)])
      show  $(\forall v \text{ bnd. } \text{Mapping.lookup } \text{bnds } v = \text{Some } \text{bnd} \implies \alpha v \in \text{set } \text{bnd})$ 
    =
       $(\forall x \text{ bnd. } \text{Mapping.lookup } (\text{Mapping.delete } v \text{ bnds}) x = \text{Some } \text{bnd} \implies$ 
 $\alpha x \in \text{set } \text{bnd})$  (is ?l = ?r)
    proof

```

```

show ?l  $\implies$  ?r by (auto simp: lookup-delete-upd)
assume ?r
show ?l
proof (intro allI impI)
  fix x bnd
  assume Mapping.lookup bnds x = Some bnd
  thus  $\alpha$  x  $\in$  set bnd using ⟨?r⟩[rule-format, of x bnd] assms
    by (cases x = v, auto simp: lookup-delete-upd)
qed
qed
next
have fidl-constraints-sat (IDL-CS vis vws cs)  $\alpha$  =
  (fidl-constraints-sat (IDL-CS vis [] []))  $\alpha$   $\wedge$ 
  fidl-constraints-sat (IDL-CS [] vws [])  $\alpha$   $\wedge$ 
  fidl-constraints-sat (IDL-CS [] [] cs)  $\alpha$ 
  by (auto simp: fidl-constraints-sat-def)
also have fidl-constraints-sat (IDL-CS vis [] [])  $\alpha$  =
  ( $\forall$  disj $\in$ set vis. fidle-sat  $\alpha$  (Var-Int disj))
  unfolding fidl-constraints-sat-def by (simp add: singleton-def)
also have ... = (( $\forall$  disj $\in$ set cvis. fidle-sat  $\alpha$  (Var-Int disj))
   $\wedge$  ( $\forall$  disj $\in$ set nvis1. fidle-sat  $\alpha$  (Var-Int disj)))
  using p1 by auto
also have ( $\forall$  disj $\in$ set cvis. fidle-sat  $\alpha$  (Var-Int disj)) = True
  using p1 cvis assms(1) by force
also have fidl-constraints-sat (IDL-CS [] vws [])  $\alpha$  = Ball (set (map
  Var-Var vws)) (fidlc-sat  $\alpha$ )
  unfolding fidl-constraints-sat-def by (simp add: singleton-def)
also have ... = ((Ball (set (map Var-Int nvis2)) (fidlc-sat  $\alpha$ )  $\wedge$  Ball
  (set (map Var-Var nvws)) (fidlc-sat  $\alpha$ ))) unfolding inst-vv ..
also have fidl-constraints-sat (IDL-CS [] [] cs)  $\alpha$  = ( $\forall$  cs $\in$ set cs. Bex
  (set cs) (fidlc-sat  $\alpha$ ))
  unfolding fidl-constraints-sat-def by simp
also have ... = ( $\forall$  cs $\in$ set ncs. Bex (set cs) (fidlc-sat  $\alpha$ )) unfolding
  inst-cs ..
finally have id: fidl-constraints-sat (IDL-CS vis vws cs)  $\alpha$  =
  (( $\forall$  disj $\in$ set (nvis1 @ nvis2). fidle-sat  $\alpha$  (Var-Int disj))  $\wedge$ 
  ( $\forall$  a $\in$ set (map Var-Var nvws). fidle-sat  $\alpha$  a)  $\wedge$ 
  ( $\forall$  cs $\in$ set ncs. Bex (set cs) (fidlc-sat  $\alpha$ ))) by auto
show fidl-constraints-sat (IDL-CS vis vws cs)  $\alpha$  = fidl-constraints-sat
  (IDL-CS (nvis1 @ nvis2) nvws ncs)  $\alpha$ 
  unfolding id unfolding fidl-constraints-sat-def by (auto simp:
  singleton-def)
qed

note fidl = assms(5)[unfolded fidl-state.simps]

show fidl-state s V unfolding s fidl-state.simps
proof (intro conjI ballI, clarsimp)
  fix x b

```

```

      assume  $(x, b) \in \text{Mapping.entries } (\text{Mapping.delete } v \text{ bnds})$ 
      hence  $\text{Mapping.lookup } (\text{Mapping.delete } v \text{ bnds}) \ x = \text{Some } b$  by (rule
in-entriesD)
      from  $\text{this}[\text{unfolded lookup-delete-upd}]$ 
      have  $xv: x \neq v$  and  $\text{Mapping.lookup } \text{bnds } \ x = \text{Some } b$  by (cases  $x = v$ ,
auto)+
      hence  $(x, b) \in \text{Mapping.entries } \text{bnds}$  by (intro in-entriesI)
      with  $\text{fidl } xv$  show  $\text{distinct } b \wedge (x \in V \vee b \neq [])$  by auto
next
let ?VI =  $\lambda \text{ vi. } \bigcup_{x \in \text{set } \text{vi.}} \text{set } (\text{fidlc-vars } (\text{Var-Int } x))$ 
let ?VV =  $\lambda \text{ vv. } \bigcup_{x \in \text{set } \text{vv.}} \text{set } (\text{fidlc-vars } (\text{Var-Var } x))$ 
let ?CS =  $\lambda \text{ cs. } \bigcup_{x \in \text{set } \text{cs.}} \bigcup_{a \in \text{set } x.} \text{set } (\text{fidlc-vars } a)$ 
have  $\text{fidl-flat-vs } (\text{IDL-CS } (\text{nvis1 } @ \text{nvis2}) \ \text{nvws } \ \text{ncs}) =$ 
 $?\text{VI } \text{nvis1} \cup (?\text{VI } \text{nvis2} \cup ?\text{VV } \text{nvws}) \cup ?\text{CS } \text{ncs}$  unfolding  $\text{fidl-flat-vs-def}$ 

      by (auto simp: singleton-def)
      also have  $\dots \subseteq (?\text{VI } \text{vis} - \{v\}) \cup (?\text{VV } \text{vws} - \{v\}) \cup (?\text{CS } \text{cs} - \{v\})$ 
      proof (intro Un-mono)
        show  $?\text{VI } \text{nvis1} \subseteq ?\text{VI } \text{vis} - \{v\}$  using  $p1$  by auto
        show  $?\text{VI } \text{nvis2} \cup ?\text{VV } \text{nvws} \subseteq ?\text{VV } \text{vws} - \{v\}$  using  $\text{inst-vv}(2)$  by
auto
        show  $?\text{CS } \text{ncs} \subseteq ?\text{CS } \text{cs} - \{v\}$  using  $\text{inst-cs}(2)$  by auto
      qed
      also have  $\dots \subseteq \text{fidl-flat-vs } (\text{IDL-CS } \text{vis } \text{vws } \ \text{cs}) - \{v\}$ 
      unfolding  $\text{fidl-flat-vs-def}$  by (auto simp: singleton-def)
      also have  $\dots \subseteq \text{Mapping.keys } \text{bnds} - \{v\}$ 
      using  $\text{fidl}[\text{unfolded css}]$  by auto
      also have  $\dots = \text{Mapping.keys } (\text{Mapping.delete } v \ \text{bnds})$  by simp
      finally show  $\text{fidl-flat-vs } (\text{IDL-CS } (\text{nvis1 } @ \text{nvis2}) \ \text{nvws } \ \text{ncs}) \subseteq \text{Map-}$ 
 $\text{ping.keys } (\text{Mapping.delete } v \ \text{bnds})$ 
      by auto
      show  $\text{finite } (\text{Mapping.keys } (\text{Mapping.delete } v \ \text{bnds}))$  using  $\text{fidl}$  by auto
    qed

      show  $\text{fidl-vars } s = \text{fidl-vars } (\text{IDL-State } \text{bnds } \ \text{css}) - \{v\}$  unfolding  $s$ 
 $\text{fidl-vars.simps}$  by auto

      from  $\text{assms}(2)$  have  $v: v \in \text{Mapping.keys } \text{bnds}$ 
      unfolding  $\text{keys-is-none-rep}$  by auto
      hence  $\text{sub}: \text{Mapping.keys } (\text{Mapping.delete } v \ \text{bnds}) \subset \text{Mapping.keys } \text{bnds}$ 
by auto
      from  $\text{fidl}$  have  $\text{finite } (\text{Mapping.keys } \text{bnds})$  by auto
      with  $\text{sub}$  show  $\text{fidl-size } s < \text{fidl-size } (\text{IDL-State } \text{bnds } \ \text{css})$  unfolding  $s$ 
 $\text{fidl-size.simps}$ 
      unfolding  $\text{Mapping.size-def}$  by (simp add: psubset-card-mono)
    qed
  qed
  qed
  qed

```

qed

lemma *instantiate-var-size*: **assumes** *instantiate-var* v i $s = \text{Some } s'$
shows $\text{fidl-size } s' \leq \text{fidl-size } s$
proof –
obtain $\text{bnds } \text{css}$ **where** $s: s = \text{IDL-State } \text{bnds } \text{css}$ **by** (*cases* s , *auto*)
obtain $\text{vis } \text{vws } \text{cs}$ **where** $\text{css}: \text{css} = \text{IDL-CS } \text{vis } \text{vws } \text{cs}$ **by** (*cases* css , *auto*)
note $s = s[\text{unfolded } \text{css}]$
show *?thesis* **using** *assms* **unfolding** s **by** (*auto split: if-splits option.splits simp: Let-def size-delete*)
qed

fun *fidl-cs-empty* :: $'v$ *fidl-solver-state* \Rightarrow *bool* **where**
fidl-cs-empty (*IDL-State* bnds (*IDL-CS* $\square \square \square$)) = *True*
| *fidl-cs-empty* - = *False*

lemma *fidl-cs-empty*: **assumes** *fidl-state* s $\{\}$
and *fidl-cs-empty* s
shows $\text{Ex } (\text{fidl-state-sat } s)$
proof –
obtain bnds **where** $s: s = \text{IDL-State } \text{bnds}$ (*IDL-CS* $\square \square \square$)
using *assms* **by** (*cases* s *rule: fidl-cs-empty.cases*, *auto*)
have *fidl-state-sat* s ($\lambda v. \text{hd } (\text{the } (\text{Mapping.lookup } \text{bnds } v))$)
using *assms*
unfolding s **by** (*auto simp: fidl-constraints-sat-def all-entries-eq-all-lookups*)
thus *?thesis* **by** *blast*
qed

lemma *fidl-vars*: **assumes** $\bigwedge v. v \in \text{fidl-vars } s \implies \alpha v = \beta v$
and *fidl-state* s V
shows *fidl-state-sat* s $\alpha = \text{fidl-state-sat } s$ β
proof –
obtain $\text{bnds } c$ **where** $s: s = \text{IDL-State } \text{bnds } c$ **by** (*cases* s , *auto*)
show *?thesis* **unfolding** s *fidl-state-sat.simps*
proof (*intro arg-cong2[of - - - (\wedge)] all-cong1 imp-cong refl arg-cong[of - - $\lambda x. x \in \cdot$]*)
fix v *bnd*
show *Mapping.lookup* $\text{bnds } v = \text{Some } \text{bnd} \implies \alpha v = \beta v$
by (*intro assms(1)*, *auto simp: s keys-is-none-rep*)
show *fidl-constraints-sat* c $\alpha = \text{fidl-constraints-sat } c$ β
unfolding *fidl-constraints-sat-def*
proof (*intro ball-cong refl bex-cong*)
fix a
assume $a \in \text{fidl-flat-cs } c$ $a \in a$
with *assms(2)[unfolded s fidl-state.simps fidl-flat-vs-def]*
have $\text{set } (\text{fidlc-vars } a) \subseteq \text{fidl-vars } s$ **by** (*auto simp: s*)
thus *fidlc-sat* α $a = \text{fidlc-sat } \beta$ a **using** *assms(1)* **by** (*cases* a , *auto*)
qed
qed

qed

```

fun clean-bnds :: 'v fidl-solver-state  $\Rightarrow$  'v list  $\Rightarrow$  'v fidl-solver-state + bool where
  clean-bnds s [] = (if fidl-cs-empty s then Inr True else Inl s)
| clean-bnds (IDL-State bnds c) (v # vs) = (case Mapping.lookup bnds v of
  None  $\Rightarrow$  clean-bnds (IDL-State bnds c) vs
| Some ints  $\Rightarrow$  (case ints of
  []  $\Rightarrow$  Inr False
| [i]  $\Rightarrow$  (case instantiate-var v i (IDL-State bnds c) of None  $\Rightarrow$  Inr False
  | Some s  $\Rightarrow$  clean-bnds s vs)
| -  $\Rightarrow$  clean-bnds (IDL-State bnds c) vs
))

```

lemma clean-bnds: **assumes** fidl-state s (set vs)

and clean-bnds s vs = res

shows res = Inr b \Longrightarrow b = (Ex (fidl-state-sat s))

res = Inl s' \Longrightarrow fidl-state s' {} \wedge Ex (fidl-state-sat s') = Ex (fidl-state-sat s)

proof (atomize(full), insert assms, induct vs arbitrary: s)

case (Nil s)

show ?case

proof (cases fidl-cs-empty s)

case True

with fidl-cs-empty[of s] Nil **show** ?thesis **by** auto

next

case False

thus ?thesis **using** Nil **by** auto

qed

next

case (Cons v vs s)

obtain bnds cs **where** s: s = IDL-State bnds cs **by** (cases s, auto)

note fidl = Cons(2)

note res = Cons(3)[symmetric, unfolded s clean-bnds.simps]

show ?case

proof (cases Mapping.lookup bnds v)

case None

from res[unfolded this]

have res: clean-bnds s vs = res **by** (auto simp: s)

from None fidl **have** fidl-state s (set vs)

unfolding s fidl-state.simps all-entries-eq-all-lookups **by** force

note IH = Cons(1)[OF this res]

thus ?thesis .

next

case (Some ints)

note res = res[unfolded Some option.simps]

show ?thesis

proof (cases ints)

case Nil

with res **have** res: res = Inr False **by** auto

```

have  $\neg$  fidl-state-sat s  $\alpha$  for  $\alpha$  unfolding s using Some[unfolded Nil]
  unfolding fidl-state-sat.simps all-entries-eq-all-lookups by force
with res show ?thesis by auto
next
case ints: (Cons i nums)
show ?thesis
proof (cases nums = [])
  case True
  with ints have ints: ints = [i] by auto
  note res = res[unfolded ints list.simps]
  note Some = Some[unfolded ints]
  have ex:  $\exists x$ . (fidl-state-sat s) = ( $\exists \alpha$ . fidl-state-sat s  $\alpha \wedge \alpha v = i$ )
    unfolding s fidl-state-sat.simps all-entries-eq-all-lookups using Some
    by (intro ex-cong1, force)
  show ?thesis
  proof (cases instantiate-var v i (IDL-State bnds cs))
    case None
    with res have res = Inr False by auto
    moreover have  $\alpha v = i \implies \neg$  fidl-state-sat s  $\alpha$  for  $\alpha$ 
      using instantiate-var(1)[OF - Some - None, folded s, of  $\alpha$  set vs] fidl
  by auto
  with ex have  $\neg$  ( $\exists x$ . (fidl-state-sat s)) by auto
  ultimately show ?thesis by auto
next
case s'': (Some s'')
with res have res: clean-bnds s'' vs = res by auto
from instantiate-var(2)[OF - Some - s'' - refl, folded s, of - set vs]
  have step:  $\alpha v = i \implies$  fidl-state s'' (set vs)  $\wedge$  fidl-state-sat s  $\alpha$  =
fidl-state-sat s''  $\alpha$ 
   $\wedge$  fidl-vars s'' = fidl-vars s - {v} for  $\alpha$ 
  using fidl by auto
from step[of  $\lambda - . i$ ] have fidl: fidl-state s'' (set vs)
  and vars: fidl-vars s'' = fidl-vars s - {v} by auto
have ex2: ( $\exists a$ . fidl-state-sat s'' a) = ( $\exists a$ . fidl-state-sat s a) (is ?l = ?r)
proof
  assume ?l
  then obtain  $\alpha$  where fidl-state-sat s''  $\alpha$  by auto
  with fidl-vars[of s'', unfolded vars, OF - fidl, of  $\alpha$ ]
  have fidl-state-sat s'' ( $\alpha (v := i)$ ) by simp
  with step[of  $\alpha (v := i)$ ] show ?r by auto
next
  assume ?r
  with ex obtain  $\alpha$  where fidl-state-sat s  $\alpha$  and  $\alpha v = i$  by auto
  with step[of  $\alpha$ ] show ?l by auto
qed
from step[of  $\lambda - . i$ ] have fidl: fidl-state s'' (set vs) by auto
note IH = Cons(1)[OF this res]
thus ?thesis unfolding ex2 by auto
qed

```

```

next
  case False
  with res have res: clean-bnds s vs = res unfolding ints s by (cases nums,
auto)
    from fidl have fidl: fidl-state s (set vs) unfolding s fidl-state.simps
all-entries-eq-all-lookups
    using Some[unfolded ints] by force
    from Cons(1)[OF fidl res] show ?thesis .
  qed
qed
qed
qed

lemma clean-bnds-size: assumes clean-bnds s xs = Inl s'
  shows fidl-size s' ≤ fidl-size s
  using assms
proof (induct s xs arbitrary: s' rule: clean-bnds.induct)
  case 1
  thus ?case by (auto split: if-splits)
next
  case (2 bnds c v vs s')
  show ?case
  proof (rule ccontr)
    assume not: ¬ ?thesis
    note res = 2(4)[simplified]
    from not 2 obtain xs where look: Mapping.lookup bnds v = Some xs by (auto
split: option.splits)
    note res = res[unfolded this, simplified]
    then obtain i ys where xs: xs = i # ys by (cases xs, auto)
    note res = res[unfolded this, simplified]
    note look = look[unfolded xs]
    note IH = 2(2-3)[OF look refl]
    show False
  proof (cases ys)
    case (Cons j zs)
      from xs look IH(2)[OF Cons, of s'] res not show False unfolding Cons
list.simps by auto
    next
      case Nil
      with res obtain s2 where inst: instantiate-var v i (IDL-State bnds c) =
Some s2
      by (auto split: option.splits)
      from res[unfolded Nil inst]
      have res: clean-bnds s2 vs = Inl s' by auto
      from IH(1)[OF Nil inst res] not res look instantiate-var-size[OF inst]
      show False using Nil by auto
    qed
  qed
qed

```

definition *fidl-init* :: *bool* \Rightarrow $((v \times s) \times \text{int}) \text{ list} \Rightarrow ((v \times s) \times v \times s) \text{ list list}$
 $\Rightarrow (v \times s) \text{ fidl-solver-state} + \text{bool}$ **where**
fidl-init sym-break *bnds* *diffs* = (let
scs = *IDL-CS* [] [] (map (map *Var-Var*) *diffs*)
in (if *sym-break* then
(let
sToV = *Mapping.of-alist* (map ($\lambda (vs,b). (\text{snd } vs, vs)$) *bnds*);
sorts = *remdups* (map (*snd o fst*) *bnds*);
chosenVs = map (*the o Mapping.lookup sToV*) *sorts*;
sbnds = *Mapping.of-alist* (map ($\lambda (vs,b). (vs, \text{if } \text{Mapping.lookup } sToV (\text{snd } vs) = \text{Some } vs \text{ then } [b] \text{ else } [0..b])$) *bnds*)
in *clean-bnds* (*IDL-State sbnds scs*) *chosenVs*)
else *Inl* (*IDL-State* (*Mapping.of-alist* (map (*map-prod id* ($\lambda b. [0..b]$)) *bnds*))
scs)))

lemma *mapping-of-alist-subset*: *Mapping.entries* (*Mapping.of-alist* *xs*) \subseteq *set* *xs*
by (*metis in-entriesD lookup-of-alist map-of-SomeD subrelI*)

term *fidl-init*

lemma *fidl-init*: **assumes** *fidl-init sym-break* *bnds* *diffs* = *res*

and *fidl-input* $((bnds, \text{diffs}) :: (v, s) \text{fidl-input})$

shows *res* = *Inl state* \Longrightarrow *fidl-state* *state* {} \wedge *fidl-solvable* (*bnds*, *diffs*) = *Ex* (*fidl-state-sat* *state*)

res = *Inr b* \Longrightarrow *b* = *fidl-solvable* (*bnds*, *diffs*)

proof –

let *?br* = *sym-break*

define *sToV* **where** *sToV* = *Mapping.of-alist* (map ($\lambda (vs,b). (\text{snd } vs, vs)$) *bnds*)

define *sorts* **where** *sorts* = *remdups* (map (*snd o fst*) *bnds*)

define *chosenVs* **where** *chosenVs* = (if *?br* then map (*the o Mapping.lookup sToV*) *sorts* else [])

define *bnd* **where** *bnd* *vs* *b* = (if *?br* then (if *Mapping.lookup sToV* (*snd* *vs*) = *Some* *vs* then [*b*] else [*0..b*]) else [*0..b*]) **for** *vs* *b*

define *sbnds* **where** *sbnds* = *Mapping.of-alist* (map ($\lambda (vs,b). (vs, \text{bnd } vs \text{ } b)$) *bnds*)

define *scs* **where** *scs* = *IDL-CS* [] [] (map (map *Var-Var*) *diffs*)

define *state1* **where** *state1* = *IDL-State* *sbnds* *scs*

have *res*: *res* = (if *?br* then *clean-bnds* *state1* *chosenVs* else *Inl* *state1*)

unfolding *assms*(1)[*symmetric*] *fidl-init-def* *Let-def* *state1-def* *scs-def* *sbnds-def* *chosenVs-def* *sorts-def* *sToV-def* *bnd-def*

by (*cases* *?br*, *auto* *intro!*: *arg-cong*[*of - - Mapping.of-alist*] *map-cong*)

note *fidl* = *assms*(2)[*unfolded* *fidl-input-def* *split*]

have *flatten*: *fidl-flat-cs* *scs* = *set* (map (*set o map* *Var-Var*) *diffs*) **unfolding** *scs-def* **by** *auto*

have *keys*: *Mapping.keys* *sbnds* = *fst* ‘ *set* *bnds* **unfolding** *sbnds-def* *keys-of-alist* **by** *force*

have *map-fst*: map *fst* (map ($\lambda (vs,b). (vs, \text{bnd } vs \text{ } b)$) *bnds*) = map *fst* *bnds* **by**

```

(induct bnds, auto)
  have entries: Mapping.entries sbnds = ( $\lambda (v,b). (v, \text{bnd } v \ b)$ ) ‘ set bnds
  unfolding sbnds-def
  apply (subst entries-of-alist)
  subgoal unfolding map-fst using fidl by blast
  subgoal by (induct bnds, auto)
  done

  have state1: fidl-state state1 (set chosenVs)
  unfolding state1-def fidl-state.simps fidl-flat-vs-def flatten keys set-map entries
  proof (intro conjI subsetI ballI)
    fix x
    assume x  $\in$  ( $\bigcup c \in (\text{set} \circ \text{map } \text{Var-Var})$  ‘ set diffs.  $\bigcup a \in c. \text{set } (\text{fidlc-vars } a)$ )
    from this
    obtain v w where vw:  $(v,w) \in \text{set } (\text{concat } \text{diffs})$  and x:  $x \in \text{set } (\text{fidlc-vars } (\text{Var-Var } (v,w)))$  by force
    from vw have  $\{v,w\} \subseteq \text{fst}$  ‘ set bnds using fidl by blast
    thus  $x \in \text{fst}$  ‘ set bnds using x by auto
  next
    fix xb
    assume xb  $\in$  ( $\lambda(v, b). (v, \text{bnd } v \ b)$ ) ‘ set bnds
    then obtain x b where mem:  $(x,b) \in \text{set } \text{bnds}$  and xb:  $xb = (x, \text{bnd } x \ b)$  by
  auto
    from fidl mem have  $0 \leq b$  by blast
    thus case xb of  $(v, \text{ints}) \Rightarrow \text{distinct } \text{ints} \wedge (v \in \text{set } \text{chosenVs} \vee \text{ints} \neq [])$ 
  unfolding xb split
    by (auto simp: bnd-def)
  qed (insert fidl, auto)

  have main: fidl-solvable (bnds, diffs) = Ex (fidl-state-sat state1) (is ?l = ?r)
  proof
    assume ?r
    then obtain  $\alpha$  where fidl-state-sat state1  $\alpha$  by auto
    note sat = this[unfolded state1-def fidl-state-sat.simps]
    show ?l unfolding fidl-solvable-def split
    proof (intro exI[of -] conjI ballI)
      fix c
      assume c  $\in$  set diffs
      with flatten[unfolded set-map]
      have  $(\text{set} \circ \text{map } \text{Var-Var}) \ c \in \text{fidl-flat-cs } \text{scs}$  by auto
      with sat[THEN conjunct2, unfolded fidl-constraints-sat-def, rule-format, OF
  this]
      show  $\exists (v, w) \in \text{set } c. \alpha \ v \neq \alpha \ w$  by force
    next
      fix vb
      assume vb  $\in$  set bnds
      then obtain v b where vb:  $vb = (v,b)$  and mem:  $(v,b) \in \text{set } \text{bnds}$ 
      by (metis surj-pair)

```

```

    with sat[THEN conjunct1, folded all-entries-eq-all-lookups, unfolded entries]
    have sat:  $\alpha v \in \text{set } (\text{bnd } v \ b)$  by auto
    from fidl mem have b:  $b \geq 0$  by blast
    from sat b show case vb of  $(v, b) \Rightarrow 0 \leq \alpha v \wedge \alpha v \leq b$  unfolding vb split
bnd-def
    by (auto split: if-splits)
qed
next

    assume ?l
    from this[unfolded fidl-solvable-def split]
    obtain  $\alpha$  where sat-bnds:  $\bigwedge v \ b. (v, b) \in \text{set } \text{bnds} \Longrightarrow 0 \leq \alpha v \wedge \alpha v \leq b$ 
and
    sat-diffs:  $\bigwedge c. c \in \text{set } \text{diffs} \Longrightarrow \exists (v, w) \in \text{set } c. \alpha v \neq \alpha w$  by auto

    define vs ::  $'s \Rightarrow ('v \times 's)$  where vs s = the (Mapping.lookup sToV s) for s

    define sv $\alpha$  ::  $'s \Rightarrow \text{int}$  where sv $\alpha$  s =  $\alpha (vs \ s)$  for s
    define svb ::  $'s \Rightarrow \text{int}$  where svb s = the (map-of bnds (vs s)) for s
    define  $\beta$  where  $\beta = (\lambda (x, s). \text{if } \alpha (x, s) = \text{sv}\alpha \ s \ \text{then } \text{svb } \ s \ \text{else if } \alpha (x, s) = \text{svb } \ s \ \text{then } \text{sv}\alpha \ s \ \text{else } \alpha (x, s))$ 

    have  $\beta$ -to- $\alpha$ :  $\text{snd } v = \text{snd } w \Longrightarrow \beta v = \beta w \longleftrightarrow \alpha v = \alpha w$  for v w unfolding
 $\beta$ -def
    by (cases v; cases w; auto)

    show ?r unfolding state1-def fidl-state-sat.simps fidl-constraints-sat-def flatten
set-map
    all-entries-eq-all-lookups[symmetric] entries
    proof (intro exI[of -] conjI allI impI ballI, clarsimp)
    fix disj
    assume disj  $\in (\text{set } \circ \text{map } \text{Var-Var}) \ ' \ \text{set } \text{diffs}$ 
    then obtain c where c:  $c \in \text{set } \text{diffs}$  and disj:  $\text{disj} = (\text{set } \circ \text{map } \text{Var-Var})$ 
c by auto
    from sat-diffs[OF c] obtain v w where vw:  $(v, w) \in \text{set } c$  and diff:  $\alpha v \neq \alpha w$ 
w by auto
    from c vw have  $(v, w) \in \text{set } (\text{concat } \text{diffs})$  by auto
    with fidl have  $\text{snd } v = \text{snd } w$  by blast
    with diff  $\beta$ -to- $\alpha$  have diff:  $\beta v \neq \beta w$  by auto
    from vw disj have Var-Var  $(v, w) \in \text{disj}$  by auto
    with diff show Bex disj (fidlc-sat  $\beta$ ) by force
next
    fix v s b
    assume mem:  $((v, s), b) \in \text{set } \text{bnds}$ 
    with fidl have b:  $b \geq 0$  by force
    have snd:  $\text{snd } (v, s) = s$  by simp
    from mem have  $s \in \text{Mapping.keys } sToV$  unfolding sToV-def keys-of-alist
by force
    then obtain v' where v':  $\text{Mapping.lookup } sToV \ s = \text{Some } v'$  by (meson

```

```

in-keysD)
  hence  $(s, v') \in \text{Mapping.entries } sToV$  by (rule in-entriesI)
  from set-mp[OF mapping-of-alist-subset this[unfolded sToV-def]] obtain  $b'$ 
    where  $vb': (v', b') \in \text{set bnds}$  and  $snd\ v': snd\ v' = s$  by auto
  with fidl mem snd have  $b' = b$  by blast
  with  $vb'$  have  $vb': (v', b) \in \text{set bnds}$  by auto
  from  $v'$  have  $vsv': vs\ s = v'$  unfolding vs-def by simp
  have  $svbs: svb\ s = b$  unfolding svb-def  $vsv'$  using  $vb'$  fidl[THEN conjunct1]
by simp
  show  $\beta\ (v, s) \in \text{set } (bnd\ (v, s)\ b)$ 
  proof (cases Mapping.lookup sToV s = Some (v, s))
    case True
      hence  $vss: vs\ s = (v, s)$  unfolding vs-def by simp
      hence  $\beta\ (v, s) = svb\ s$  unfolding  $\beta$ -def split  $sv\alpha$ -def by simp
      also have  $svb\ s = b$  by fact
      finally show ?thesis unfolding bnd-def using True b by simp
    next
      case False
      hence  $set: \text{set } (bnd\ (v, s)\ b) = \{0..b\}$ 
        unfolding bnd-def by auto
      from sat-bnds[OF  $vb'$ ]
      have  $sv\alpha\ s \in \{0..b\}$  unfolding  $sv\alpha$ -def  $vsv'$  by auto
      moreover
      from sat-bnds[OF mem]
      have  $\alpha\ (v, s) \in \{0..b\}$  by auto
      moreover
      from b svbs have  $svb\ s \in \{0..b\}$  by auto
      ultimately
      show ?thesis unfolding set  $\beta$ -def split by auto
  qed
qed
qed

have  $chosen: \neg\ ?br \implies \text{set } chosenVs = \{\}$  by (auto simp: chosenVs-def)
note clean = clean-bnds[OF state1 refl]
show  $res = Inl\ state \implies res = Inl\ state \implies \text{fidl-state } state\ \{\} \wedge \text{fidl-solvable}$ 
 $(bnds, diffs) = Ex\ (\text{fidl-state-sat } state)$ 
  using res clean(2)[of state] state1 chosen unfolding main by (cases ?br; force)
show  $res = Inr\ b \implies b = \text{fidl-solvable } (bnds, diffs)$ 
  using res clean(1)[of b] unfolding main by (cases ?br; force)
qed

```

definition deduction-step :: 'v fidl-solver-state \Rightarrow 'v fidl-solver-state + bool **where**
 deduction-step s = (case fidl-delete-vi s of
 (s1, vs) \Rightarrow (case clean-bnds s1 vs of
 Inr b \Rightarrow Inr b

```

| Inl s2 ⇒ (case fidl-restructure s2 of
  None ⇒ Inr False
| Some s3 ⇒ Inl s3)))

```

```

lemma deduction-step: assumes fidl: fidl-state s {}
and res: deduction-step s = res
shows res = Inr b ⇒ b = (Ex (fidl-state-sat s))
  res = Inl s' ⇒ fidl-state s' {} ∧ Ex (fidl-state-sat s') = Ex (fidl-state-sat s)
proof (atomize(full), goal-cases)
  case 1
  obtain s1 vs where del: fidl-delete-vi s = (s1,vs) by force
  note res = res[unfolded deduction-step-def del split]
  from fidl-delete-vi[OF del fidl]
  have fidl: fidl-state s1 (set vs) and
    eq: Ex (fidl-state-sat s) = Ex (fidl-state-sat s1) by auto
  show ?case
  proof (cases clean-bnds s1 vs)
    case (Inr r)
    from clean-bnds(1)[OF fidl Inr refl] Inr res eq show ?thesis by auto
  next
    case (Inl s2)
    from clean-bnds(2)[OF fidl Inl refl] eq
    have fidl: fidl-state s2 {} and eq: Ex (fidl-state-sat s) = Ex (fidl-state-sat s2)
      by auto
    note res = res[unfolded Inl sum.simps]
    show ?thesis
    proof (cases fidl-restructure s2)
      case None
      from fidl-restructure(1)[OF None] eq res None show ?thesis by auto
    next
      case (Some s3)
      from fidl-restructure(2-)[OF Some] eq fidl Some res show ?thesis by auto
    qed
  qed
qed

```

```

lemma deduction-step-size: assumes deduction-step s = Inl s'
shows fidl-size s' ≤ fidl-size s
using assms unfolding deduction-step-def
by (auto split: prod.splits sum.splits option.splits
  dest!: clean-bnds-size fidl-delete-vi-size fidl-restructure)

```

```

fun deduction-steps-main :: nat ⇒ 'v fidl-solver-state ⇒ 'v fidl-solver-state + bool
where
  deduction-steps-main n s = (case deduction-step s of
    Inr b ⇒ Inr b
  | Inl s' ⇒ let n' = fidl-size s' in if n' < n then deduction-steps-main n' s' else
    Inl s')

```

```

declare deduction-steps-main.simps[simp del]

definition deduction-steps s = deduction-steps-main (fidl-size s) s

lemma deduction-steps: assumes fidl: fidl-state s {}
  and res: deduction-steps s = res
shows res = Inr b  $\implies$  b = (Ex (fidl-state-sat s))
  res = Inl s'  $\implies$  fidl-state s' {}  $\wedge$  Ex (fidl-state-sat s') = Ex (fidl-state-sat s)
proof (atomize(full), goal-cases)
  case 1
  from assms show ?case unfolding deduction-steps-def
  proof (induct s rule: measure-induct[of fidl-size])
    case (1 s)
    note res = 1(3)[unfolded deduction-steps-main.simps[of - s]]
    note IH = 1(1)
    note fidl = 1(2)
    note ded = deduction-step[OF fidl refl]
    show ?case
    proof (cases deduction-step s)
      case (Inr b)
      from ded(1)[OF this] res Inr show ?thesis by auto
    next
      case (Inl s1)
      note res = res[unfolded Inl sum.simps Let-def]
      from ded(2)[OF Inl]
      have fidl: fidl-state s1 {}
      and eq: Ex (fidl-state-sat s) = Ex (fidl-state-sat s1) by auto
      show ?thesis
      proof (cases fidl-size s1 < fidl-size s)
        case True
        with res have res: deduction-steps-main (fidl-size s1) s1 = res by auto
        from IH[rule-format, OF True fidl res]
        show ?thesis unfolding eq by auto
      next
        case False
        with res eq fidl show ?thesis by auto
      qed
    qed
  qed
qed
qed

lemma deduction-steps-size: assumes deduction-steps s = Inl s'
  shows fidl-size s'  $\leq$  fidl-size s
  using assms unfolding deduction-steps-def
proof (induct s arbitrary: s' rule: measure-induct[of fidl-size])
  case (1 s s')
  thus ?case unfolding deduction-steps-main.simps[of - s] Let-def
  by (auto split: sum.splits if-splits dest: deduction-step-size)
qed

```

```

fun fidl-var-int where
  fidl-var-int (Var-Int vi) = (fst vi, Some (snd vi))
| fidl-var-int (Var-Var vw) = (fst vw, None)

fun fidl-find-var :: 'v fidl-solver-state ⇒ ('v × int option) + bool where
  fidl-find-var (IDL-State bnds (IDL-CS vis vws cs)) = (
    case vis of
      vi # vis2 ⇒ Inl (map-prod id Some vi)
    | - ⇒ (case vws of vw # vws2 ⇒ Inl (fst vw, None)
    | - ⇒ (case cs of [] ⇒ Inr True
    | c # cs2 ⇒ (case c of [] ⇒ Inr False | a # as ⇒ Inl (fidl-var-int a))))))

fun reorder :: int option ⇒ int list ⇒ int list where
  reorder None xs = xs
| reorder (Some i) xs = (case span ((<) i) xs of
  (bef, j # aft) ⇒ if i = j then bef @ aft @ [j] else xs
| - ⇒ xs)

lemma set-reorder[simp]: set (reorder io xs) = set xs
proof (cases io)
  case (Some i)
    obtain bef aft where spa: span ((<) i) xs = (bef, aft) by force
    from this[symmetric, unfolded span]
    have set xs = set bef ∪ set aft by simp
      (metis set-append take-While-drop-While-id)
    thus ?thesis unfolding Some reorder.simps spa
      by (auto split: list.splits)
qed auto

definition finite-mapping m = finite (Mapping.keys m)

lemma finite-mapping-code[code]: finite-mapping (Mapping m) = True
  unfolding finite-mapping-def
  by (simp add: keys-Mapping)

function fidl-main-solver :: 'v fidl-solver-state ⇒ bool where
  fidl-main-solver s = (case fidl-restructure s of None ⇒ False
  | Some s1 ⇒ (case deduction-steps s1 of
    Inr b ⇒ b
  | Inl s2 ⇒ (case fidl-find-var s2 of
    Inr b ⇒ b
  | Inl (v, io) ⇒ (case s2 of IDL-State bnds cs ⇒
    if finite-mapping bnds then
      (case Mapping.lookup bnds v of
        Some ints ⇒ Bex (set (reorder io ints)) (λ i. map-option fidl-main-solver
      (instantiate-var v i s2) = Some True)
        ) else Code.abort (STR "infinite bnds are not allowed") (λ -. True))))))

```

by *pat-completeness auto*

termination

proof (*standard, rule wf-measure[of fidl-size], goal-cases*)

case (*1 s s1 s2 vo v io bnds cs ints i s3*)

from 1 have *fin: finite (Mapping.keys bnds)*

by (*auto simp: finite-mapping-def*)

from 1 have *v ∈ Mapping.keys bnds* by (*auto simp add: keys-dom-lookup*)

hence *Mapping.keys (Mapping.delete v bnds) ⊂ Mapping.keys bnds* by *auto*

with 1 *fin* have *lt: Mapping.size (Mapping.delete v bnds) < Mapping.size bnds*

unfolding *Mapping.size-def* by (*simp add: psubset-card-mono*)

have *fidl-size s3 = Mapping.size (Mapping.delete v bnds)*

using 1(9) unfolding 1(5)

by (*cases cs, auto split: if-splits option.splits simp: Let-def*)

also have *... < fidl-size s2* using *lt* unfolding 1 by *simp*

also have *... ≤ fidl-size s1* using *deduction-steps-size[OF 1(2)]*.

also have *... ≤ fidl-size s* using *fidl-restructure(2-)[OF 1(1)]* by *auto*

finally show *?case* by *auto*

qed

declare *fidl-main-solver.simps[simp del]*

lemma *fidl-main-solver: assumes fidl-state s {}*

shows *fidl-main-solver s = Ex (fidl-state-sat s)*

using *assms*

proof (*induct s rule: fidl-main-solver.induct*)

case (*1 s*)

note *fidl = 1(2)*

note *IH = 1(1)*

note *res = fidl-main-solver.simps[of s]*

show *?case*

proof (*cases fidl-restructure s*)

case *None*

with *res fidl-restructure[of s]* show *?thesis* by *auto*

next

case (*Some s1*)

note *res = res[unfolded Some option.simps]*

note *IH = IH[OF Some]*

from *fidl fidl-restructure(2-)[OF Some]*

have *eq: fidl-state-sat s = fidl-state-sat s1*

and *fidl: fidl-state s1 {}* by *auto*

show *?thesis*

proof (*cases deduction-steps s1*)

case (*Inr b*)

from *deduction-steps(1)[OF fidl Inr refl]* *res[unfolded Inr]* *eq* show *?thesis*

by *auto*

next

case (*Inl s2*)

note *res = res[unfolded Inl sum.simps]*

```

note IH = IH[OF Inl]
from deduction-steps(2)[OF fidl Inl refl] eq
have fidl: fidl-state s2 {} and eq: Ex (fidl-state-sat s) = Ex (fidl-state-sat
s2) by auto
obtain bnds diffs where s2: s2 = IDL-State bnds diffs by (cases s2, auto)
obtain vis vws cs where diffs: diffs = IDL-CS vis vws cs by (cases diffs,
auto)
show ?thesis
proof (cases fidl-find-var s2)
case (Inr b)
with res have res: fidl-main-solver s = b by auto
show ?thesis
proof (cases b)
case True
with Inr have fidl-cs-empty s2 unfolding s2 diffs
by (auto split: list.splits)
from fidl-cs-empty[OF fidl this] res True show ?thesis using eq by auto
next
case False
with Inr obtain cs' where cs = [] # cs' unfolding s2 diffs
by (auto split: list.splits)
hence ¬ Ex (fidl-state-sat s2)
unfolding s2 diffs by (auto simp: fidl-constraints-sat-def)
with eq False res show ?thesis by auto
qed
next
case (Inl vo)
then obtain v io where Inl: fidl-find-var s2 = Inl (v,io) by (cases vo,
auto)
from fidl[unfolded s2]
have fin: finite (Mapping.keys bnds) by auto
from Inl have v ∈ fidl-flat-vs diffs unfolding s2 diffs fidl-flat-vs-def
fidl-flat-cs.simps singleton-def
by (cases hd (hd cs), auto split: list.splits)
with fidl[unfolded s2] have v: v ∈ Mapping.keys bnds by auto
from fin have finm: finite-mapping bnds unfolding finite-mapping-def by
auto
hence finite-mapping bnds = True by simp
note res = res[unfolded Inl sum.simps split, unfolded s2 fidl-solver-state.simps,
folded s2, unfolded this if-True]
from v obtain ints where look: Mapping.lookup bnds v = Some ints
by (meson in-keysD)
note IH = IH[OF Inl refl s2 finm look, unfolded set-reorder]
note res = res[unfolded look option.simps split]
from fidl have fidlv: fidl-state s2 (insert v {}) unfolding s2 by auto
show ?thesis
proof
assume fidl-main-solver s
from this[unfolded res split]

```

```

obtain  $i\ s3$  where  $i: i \in \text{set ints}$ 
  and  $inst: \text{instantiate-var } v\ i\ s2 = \text{Some } s3$ 
  and  $res: \text{fidl-main-solver } s3 = \text{True}$  by auto
  from  $\text{instantiate-var}(2)[OF - \text{look } i, \text{of} - \text{diffs}, \text{folded } s2, OF - \text{inst } fidlv$ 
 $\text{refl}]$ 
  have  $inst': \alpha\ v = i \implies \text{fidl-state } s3\ \{\} \wedge \text{fidl-state-sat } s2\ \alpha = \text{fidl-state-sat}$ 
 $s3\ \alpha \wedge v \notin \text{fidl-vars } s3$ 
  for  $\alpha$  by auto
  from  $inst'[of\ \lambda\ -. i]$  have  $fidl3: \text{fidl-state } s3\ \{\}$  and  $v: v \notin \text{fidl-vars } s3$ 
by auto
  from  $IH[OF\ i\ inst\ fidl3]$   $res$  obtain  $\alpha$  where  $\text{fidl-state-sat } s3\ \alpha$  by auto
  with  $\text{fidl-vars}[of\ s3\ \alpha, OF - fidl3]\ v$ 
  have  $sat: \text{fidl-state-sat } s3\ (\alpha\ (v := i))$  by force
  with  $inst'[of\ \alpha\ (v := i)]$  have  $Ex\ (\text{fidl-state-sat } s2)$  by auto
  with eq show  $Ex\ (\text{fidl-state-sat } s)$  by auto
next
  assume  $Ex\ (\text{fidl-state-sat } s)$ 
  with eq obtain  $\alpha$  where  $sat: \text{fidl-state-sat } s2\ \alpha$  by auto
  from  $this[\text{unfolded } s2\ \text{fidl-state-sat.simps}]$   $look$ 
  have  $mem: \alpha\ v \in \text{set ints}$  by auto
  note  $inst = \text{instantiate-var}[OF - \text{look } mem, \text{of } \alpha\ \text{diffs}, \text{folded } s2, OF\ \text{refl}$ 
 $\text{refl } fidlv]$ 
  from  $sat\ inst(1)$  obtain  $s3$  where  $instSome: \text{instantiate-var } v\ (\alpha\ v)\ s2$ 
 $= \text{Some } s3$ 
  by auto
  from  $inst(2)[OF\ instSome]$   $sat$ 
  have  $fidl3: \text{fidl-state } s3\ \{\}$  and  $sat: \text{fidl-state-sat } s3\ \alpha$  by auto
  from  $IH[OF\ mem\ instSome\ fidl3]$   $sat$ 
  have  $IH: \text{fidl-main-solver } s3 = \text{True}$  by auto
  show  $\text{fidl-main-solver } s$  unfolding  $res$  using  $mem\ instSome\ IH$  by auto
qed
qed
qed
qed
qed

```

definition $\text{inner-solver sym-break} = (\lambda\ (bnds, \text{diffs}). \text{case } \text{fidl-init sym-break } bnds\ \text{diffs}$
 $\text{of } Inl\ s \implies \text{fidl-main-solver } s \mid Inr\ b \implies b)$

lemma $\text{inner-solver: finite-idl-solver } (\text{inner-solver sym-break})$
unfolding $\text{inner-solver-def finite-idl-solver-def}$
proof ($\text{intro allI impI, goal-cases}$)
case $(1\ \text{input})$
obtain $bnds\ \text{diffs}$ **where** $\text{input: input} = (bnds, \text{diffs})$ **by force**
note $1 = 1[\text{unfolded input}]$
show $?case$
proof ($\text{cases } \text{fidl-init sym-break } bnds\ \text{diffs}$)
case $(Inr\ b)$

```

    from fidl-init(2)[OF refl 1 Inr] Inr input show ?thesis by auto
next
  case (Inl s)
  from fidl-init(1)[OF refl 1 Inl] Inl fidl-main-solver[of s] input
  show ?thesis by auto
qed
qed

```

definition *parametric-fidl-solver* :: *bool* \Rightarrow *bool* \Rightarrow ('v,'s)fidl-input \Rightarrow *bool* **where**
parametric-fidl-solver sort-pre-process sym-break = (if sort-pre-process then
fidl-pre-processor (inner-solver sym-break) else *inner-solver sym-break*)

lemma *parametric-fidl-solver: finite-idl-solver (parametric-fidl-solver sort-pp sym-break)*

unfolding *parametric-fidl-solver-def* **using** *inner-solver fidl-pre-processor* **by**
auto

definition *default-fidl-solver* = *parametric-fidl-solver True True*

lemma *default-fidl-solver: finite-idl-solver default-fidl-solver*
unfolding *default-fidl-solver-def* **by** (rule *parametric-fidl-solver*)

```

end
theory Pattern-Completeness-Improved-Algorithm
imports
  Pattern-Completeness-List
  FCF-List
  Finite-IDL-Solver
begin

```

Combining the three solvers to get the full algorithm of section 5: arbitrary-
to-fcf: *pat-complete-impl-fcf*, fcf-to-idl: *pattern-completeness-context.fcf-solver-alg*,
and idl-solver: *default-fidl-solver*

definition *decide-pat-complete* :: *-* \Rightarrow *-* \Rightarrow (('f \times 's list) \times 's)list \Rightarrow ('f,'v,'s)pat-problem-list
 \Rightarrow *bool* **where**

```

  decide-pat-complete rn rv Cs P = (let
    m = max-arity-list Cs;
    Cl = constr-list Cs;
    Cm = Mapping.of-alist Cs;
    k = compute-k-parameter P;
    (IS,CD) = compute-inf-card-sorts-bnd k Cs;
    solve = pattern-completeness-context.fcf-solver-alg (Mapping.lookup Cm) m
  Cl CD default-fidl-solver
  in pat-complete-impl-fcf m Cl ( $\lambda$  s. s  $\in$  IS) (Mapping.lookup Cm) rn rv solve
  P)

```

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  $P$ :  $\text{snd } \bigcup (\text{vars } \text{fst } \text{set } (\text{concat } (\text{concat } P))) \subseteq \text{set } (\text{sorts-of-ssig-list } Cs)$ 
and  $\text{ren}$ :  $\text{renaming-funs } rn \ rv$ 
shows  $\text{decide-pat-complete } rn \ rv \ Cs \ P \longleftrightarrow \text{pats-complete } (\text{map-of } Cs) (\text{pat-list } \text{set } P)$ 
proof –
  let  $?k = \text{compute-k-parameter } P$ 
  interpret  $\text{pattern-completeness-list } Cs \ ?k$ 
  apply  $\text{unfold-locales}$ 
  using  $\text{dist non-empty-sorts compute-k-parameter-1}$  .
  have  $\text{nemp}$ :
     $\forall f \ \tau s \ \tau \ \tau'. f : \tau s \rightarrow \tau \text{ in } \text{map-of } Cs \longrightarrow \tau' \in \text{set } \tau s \longrightarrow \neg \text{empty-sort } (\text{map-of } Cs) \ \tau'$ 
    using  $C\text{-sub-}S$  by  $(\text{auto intro! nonempty-sort})$ 
  obtain  $\text{inf } cd$  where  $\text{compute-inf-card-sorts-bnd } ?k \ Cs = (\text{inf}, cd)$  by  $\text{force}$ 
  with  $\text{compute-inf-card-sorts-bnd}(2,3)[OF \ \text{refl } \text{nemp } \text{dist } \text{this}]$ 
  have  $\text{cics}$ :  $\text{compute-inf-card-sorts-bnd } ?k \ Cs = (\text{compute-inf-sorts } Cs, \text{min } ?k \ o \ \text{card-of-sort } (\text{map-of } Cs))$ 
  by  $(\text{simp add: Compute-Nonempty-Infinite-Sorts.compute-inf-sorts}(2) \ \text{dist } \text{nemp})$ 
  have  $\text{Cm}$ :  $\text{Mapping.lookup } (\text{Mapping.of-alist } Cs) = \text{map-of } Cs$  using  $\text{dist}$ 
  using  $\text{lookup-of-alist}$  by  $\text{fastforce}$ 
  have  $\text{fcf}$ :  $\text{pattern-completeness-context.fcf-solver-alg } (\text{map-of } Cs) (\text{max-arity-list } Cs) (\text{constr-list } Cs)$ 
     $(\text{min } ?k \ o \ \text{card-of-sort } (\text{map-of } Cs)) \ \text{default-fidl-solver}$ 
     $= \text{fcf-solver-alg } \ \text{default-fidl-solver}$ 
  unfolding  $\text{compute-card-sorts}$  by  $\text{auto}$ 
show  $?thesis$ 
apply  $(\text{unfold } \text{decide-pat-complete-def } \text{Let-def } \text{case-prod-beta } \text{cics } \text{fst-conv } \text{snd-conv})$ 
unfolding  $\text{pat-complete-impl-fcf-def } \text{Cm}$ 
apply  $(\text{rule } \text{pat-complete-impl}[OF \ \text{ren } - \ \text{refl } P \ \text{refl}])$ 
by  $(\text{rule } \text{fcf-solver-alg}[OF \ \text{default-fidl-solver}, \ \text{folded } \text{fcf}])$ 
qed

export-code  $\text{decide-pat-complete}$  checking
end

```

8 Pattern-Completeness and Related Properties

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

```

theory  $\text{Pattern-Completeness}$ 
imports
   $\text{Pattern-Completeness-List}$ 
   $\text{Pattern-Completeness-Improved-Algorithm}$ 
   $\text{Show.Shows-Literal}$ 
   $\text{Certification-Monads.Check-Monad}$ 
   $\text{Sorted-Terms.Basic-Terms}$ 

```

begin

lemmas [iff del] = domIff

A pattern completeness decision procedure for a set of lhss

definition *matches* :: ('f,'v)term ⇒ ('f,'w)term ⇒ bool (**infix** <matches> 50)
where

l matches t = (∃ σ. t = l · σ)

lemma *matches-subst*: l matches t ⇒ l matches t·σ

by (auto simp: matches-def simp flip: subst-subst-compose)

definition *pat-complete-lhss* :: ('f,'s)ssig ⇒ ('f,'s)ssig ⇒ ('f,'v)term set ⇒ bool
where

pat-complete-lhss C D L = (∀ t ∈ dom $\mathcal{T}_B(C,D)$. ∃ l ∈ L. l matches t)

lemma *pat-complete-lhssD*:

assumes *comp*: *pat-complete-lhss C D L* **and** *t*: t ∈ dom $\mathcal{T}_B(C,D,\emptyset)$

shows ∃ l ∈ L. l matches t

proof –

note * = map-subst-hastype[OF sorted-map-empty, of C - - $\emptyset::\text{unit} \rightarrow \text{undefined}$]

from t **have** t.undefined ∈ dom $\mathcal{T}_B(C,D)$ (**is** ?t ∈ -)

by (auto 0 3 simp: in-dom-iff-ex-type hastype-Basic intro!:*)

from comp[unfolded pat-complete-lhss-def, rule-format, OF this]

obtain l **where** l: l ∈ L l matches ?t **by** auto

from t

have t2: ?t · undefined = t

by (simp add: in-dom-Basic-empty-subst-subst in-dom-Basic-empty-subst-id)

from l **show** ∃ l ∈ L. l matches t

apply (subst t2[symmetric])

by (force simp: matches-subst)

qed

definition *pats-of-lhss* :: (('f × 's list) × 's)list ⇒ ('f,'v)term list ⇒ ('f,'v,'s)pat-problem-list list **where**

pats-of-lhss D lhss = (let pats = [Fun f (map Var (zip [0..(f,ss),s] ← D]

in [[(pat,lhs)]. lhs ← lhss]. pat ← pats])

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

definition *decide-pat-complete-lhss* ::
 $- \Rightarrow - \Rightarrow ((f \times 's \text{ list}) \times 's \text{ list}) \Rightarrow ((f \times 's \text{ list}) \times 's \text{ list}) \Rightarrow (f, 'v) \text{ term list} \Rightarrow \text{showsl} + \text{bool}$ **where**
decide-pat-complete-lhss rn rv C D lhss = do {
 check-signatures C D;
 return (decide-pat-complete rn rv C (pats-of-lhss D lhss))
}

lemma *pats-of-lhss-vars*: **assumes** condD: $\forall x \in \text{set } D. \forall a b. (\forall x2. x \neq ((a, b), x2)) \vee (\forall x \in \text{set } b. x \in S)$

shows $\text{snd} \text{ ' } \bigcup (\text{vars} \text{ ' } \text{fst} \text{ ' } \text{set} (\text{concat} (\text{concat} (\text{pats-of-lhss } D \text{ lhss}))) \subseteq S$

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 $\forall x \in \text{set } D. \forall a b. (\forall x2. x \neq ((a, b), x2)) \vee (\forall x \in \text{set } b. x \in \text{set} (\text{sorts-of-ssig-list}$

```

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:  $\forall x \in \text{set } D. \forall a b. (\forall x^2. x \neq ((a, b), x^2)) \vee (\forall x \in \text{set } b. x \in \text{set } S)$ 
  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)) =
    ( $\forall t \in \text{dom } \mathcal{T}_B(\text{map-of } C, \text{map-of } D). \exists l \in \text{set } \text{lhss}. l \text{ matches } t$ )
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 ( $\lambda ((f,ss),s). \text{Fun } f (\text{map } \text{Var } (\text{zip } [0..<\text{length } ss] ss))$ ) D
  define P where P = map ( $\lambda \text{pat}. \text{map } (\lambda \text{lhs}. [(pat, \text{lhs})]) \text{lhss}$ ) 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 ·  $\sigma$  | pat  $\sigma$ . pat  $\in$  set pats  $\wedge$   $\sigma$  :s {x :  $\iota$  in  $\mathcal{V}$ .  $\iota \in$  set S}  $\rightarrow$   $\mathcal{T}(\text{?C})$ 

  interpret pattern-completeness-list C 2
  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 =  $\lambda t. \exists l \in \text{set } \text{lhss}. l \text{ matches } t$ 
  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  $\in$  set lhss } | pat. pat  $\in$  set

```

```

pats}
  unfolding pat-list-def P-def by (auto simp: image-comp)
  also have  $(\forall f :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in \text{set } S\} \rightarrow \mathcal{T}(\text{map-of } C).$ 
   $\forall pp \in \{\{(pat, lhs)\} \mid lhs. lhs \in \text{set } lhs\} \mid pat. pat \in \text{set } pats\}.$ 
   $\exists mp \in pp. \text{match-complete-wrt } f \text{ mp} = \text{Ball } \{pat \cdot \sigma \mid pat \sigma. pat \in \text{set } pats \wedge$ 
 $\sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in \text{set } S\} \rightarrow \mathcal{T}(?C)\} \text{ ?match-lhs (is - = Ball ?L -)}$ 
  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  $?L = \text{dom } \mathcal{T}_B(?C, ?D, \emptyset)$  (is - = ?R)
  proof
  {
    fix pat and  $\sigma$ 
    assume pat:  $pat \in \text{set } pats$  and subst:  $\sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in \text{set } S\} \rightarrow \mathcal{T}(?C)$ 
    from pat[unfolded pats-def] obtain f ss s where pat:  $pat = \text{Fun } f \text{ (map Var$ 
(zip [ $0..<\text{length } ss$ ] ss))
    and inDs:  $((f, ss), s) \in \text{set } D$  by auto
    from distD inDs have f:  $f : ss \rightarrow s$  in ?D unfolding fun-hastype-def by
simp
    {
      fix i
      assume i:  $i < \text{length } ss$ 
      hence ss ! i  $\in \text{set } ss$  by auto
      with inDs condD have ss ! i  $\in \text{set } S$  by (auto simp: S-def)
      then
      have  $\sigma (i, ss ! i) : ss ! i \text{ in } \mathcal{T}(?C)$ 
      by (auto intro!: sorted-mapD[OF subst] simp: hastype-restrict)
    } note ssigma = this
    define ts where ts =  $(\text{map } (\lambda i. \sigma (i, ss ! i)) [0..<\text{length } ss])$ 
    have ts:  $ts :_i ss \text{ in } \mathcal{T}(?C)$  unfolding list-all2-conv-all-nth ts-def using ssigma
  by auto
    have pat:  $pat \cdot \sigma = \text{Fun } f \text{ } ts$ 
    unfolding pat ts-def by (auto intro: nth-equalityI)
    from pat f ts have  $pat \cdot \sigma \in ?R$  by (auto simp: in-dom-Basic)
  }
  thus  $?L \subseteq ?R$  by auto
  {
    fix f ss s and ts
    assume f:  $f : ss \rightarrow s$  in ?D and ts:  $ts :_i ss \text{ in } \mathcal{T}(?C)$ 
    from ts have len:  $\text{length } ts = \text{length } ss$  by (metis list-all2-lengthD)
    define pat where pat =  $\text{Fun } f \text{ (map Var (zip } [0..<\text{length } ss] \text{ } ss))$ 
    from f have  $((f, ss), s) \in \text{set } D$  unfolding fun-hastype-def by (metis map-of-SomeD)
    hence pat:  $pat \in \text{set } pats$  unfolding pat-def pats-def by force
    define  $\sigma$  where  $\sigma x = (\text{case } x \text{ of } (i, s) \Rightarrow \text{if } i < \text{length } ss \wedge s = ss ! i \text{ then}$ 
ts ! i else
    (SOME t. t : s in T(?C))) for x
    have id:  $\text{Fun } f \text{ } ts = pat \cdot \sigma$  unfolding pat-def using len
  }

```

```

    by (auto intro!: nth-equalityI simp:  $\sigma$ -def)
  have  $\sigma$ :  $\sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in \text{set } S\} \rightarrow \mathcal{T}(?C)$ 
  proof (intro sorted-mapI)
    fix  $x \ \iota$ 
    assume  $x : \iota \text{ in } \{x : \iota \text{ in } \mathcal{V}. \iota \in \text{set } S\}$ 
    then have  $\iota = \text{snd } x$  and  $s : \iota \in \text{set } S$  by auto
    then obtain  $i$  where  $x = (i, \iota)$  by (cases  $x$ , auto)
    show  $\sigma \ x : \iota \text{ in } \mathcal{T}(?C)$ 
    proof (cases  $i < \text{length } ss \wedge \iota = ss ! i$ )
      case True
        hence  $id : \sigma \ x = ts ! i$  unfolding  $x \ \sigma$ -def by auto
        from  $ts$  True show ?thesis unfolding id unfolding  $x \ \text{snd-conv}$ 
          by (auto simp add: list-all2-conv-all-nth)
      next
        case False
          hence  $id : \sigma \ x = (\text{SOME } t. t : \iota \text{ in } \mathcal{T}(?C))$  unfolding  $x \ \sigma$ -def by auto
          from decide-nonempty-sorts(1)[OF  $\text{dist } C \ \text{dec}$ ]  $s$ 
          have  $\exists t. t : \iota \text{ in } \mathcal{T}(?C)$  by (auto elim!: not-empty-sortE simp:  $S$ -def)
          from someI-ex[OF this] have  $\sigma \ x : \iota \text{ in } \mathcal{T}(?C)$  unfolding id .
          thus ?thesis unfolding  $x$  by auto
    qed
  qed
  from pat id  $\sigma$ 
  have  $\text{Fun } f \ ts \in ?L$  by auto
}
thus  $?R \subseteq ?L$  by (auto simp: in-dom-Basic)
qed
finally show ?thesis .
qed

```

theorem *decide-pat-complete-lhss-fscd*:

```

  fixes  $C \ D :: ((f \times 's \ \text{list}) \times 's) \ \text{list}$  and  $lhss :: (f, 'v) \ \text{term list}$ 
  assumes decide-pat-complete-lhss-fscd  $C \ D \ lhss = \text{return } b$ 
  shows  $b = \text{pat-complete-lhss } (\text{map-of } C) (\text{map-of } D) (\text{set } lhss)$ 

```

proof –

```

  let  $?C = \text{map-of } C$ 
  let  $?D = \text{map-of } D$ 
  define  $S$  where  $S = \text{sorts-of-ssig-list } C$ 
  define  $P$  where  $P = \text{pats-of-lhss } D \ lhss$ 
  have  $\text{sig} : \text{isOK}(\text{check-signatures } C \ D)$ 
    and  $b : b = \text{decide-pat-complete-fscd } C \ P$ 
  using assms
  apply (unfold decide-pat-complete-lhss-fscd-def)
  apply (unfold Let-def P-def[symmetric] S-def[symmetric])
  by auto
  note  $*$  = check-signatures[OF  $\text{sig}$ ]
  note  $\text{dist } C = *(1)$  note  $\text{dist } D = *(2)$  note  $\text{cond } D = *(3)$  note  $\text{dec} = *(4)$ 
  interpret pattern-completeness-list  $C \ 2$ 
  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-fscd[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 ... = ( $\forall t \in \text{dom } \mathcal{T}_B(?C, ?D). \exists l \in \text{set } lhss. l \text{ matches } t$ ) 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 2
  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 ∈ dom  $\mathcal{T}_B(?C, ?D)$ . ∃ 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:
fixes C D :: (('f × 's list) × 's) list and lhss :: ('f, 'v) term list
assumes decide-pat-complete-lhss rn rv C D lhss = return b
and ren: renaming-funs rn rv
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 rn rv 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 2
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] - ren])
apply (unfold P-def)
apply (rule pats-of-lhss-vars[OF condD[unfolded P-def S-def]])
done

```

also have ... = $(\forall t \in \text{dom } \mathcal{T}_B(?C, ?D). \exists l \in \text{set } lhss. l \text{ matches } t)$ **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* :: $(f, 's)ssig \Rightarrow (f, 's)ssig \Rightarrow (f, 'v)\text{term set} \Rightarrow \text{bool}$ **where**
strong-quasi-reducible C D L =
 $(\forall t \in \text{dom } \mathcal{T}_B(C, D). \exists ti \in \text{set } (t \# \text{args } t). \exists l \in L. l \text{ matches } ti)$

definition *term-and-args* :: $f \Rightarrow (f, 'v)\text{term list} \Rightarrow (f, 'v)\text{term list}$ **where**
term-and-args f ts = Fun f ts # ts

definition *decide-strong-quasi-reducible* ::
 $- \Rightarrow - \Rightarrow ((f \times 's \text{ list}) \times 's)\text{list} \Rightarrow ((f \times 's \text{ list}) \times 's)\text{list} \Rightarrow (f, 'v)\text{term list} \Rightarrow \text{shows}l + \text{bool}$ **where**
decide-strong-quasi-reducible rn rv C D lhss = do {
check-signatures C D;
let pats = map ($\lambda ((f, ss), s). \text{term-and-args } f (\text{map Var } (\text{zip } [0..<\text{length } ss] ss))$)
D;
let P = map (List.maps ($\lambda \text{pat}. \text{map } (\lambda \text{lhs}. [(pat, lhs)]) \text{lhss}$)) pats;
return (decide-pat-complete rn rv C P)
}

lemma *decide-strong-quasi-reducible*:

fixes *C D* :: $((f \times 's \text{ list}) \times 's)\text{list}$ **and** *lhss* :: $(f, 'v)\text{term list}$
assumes *decide-strong-quasi-reducible rn rv C D lhss = return b*
and *ren: renaming-funs rn rv*
shows $b = \text{strong-quasi-reducible } (\text{map-of } C) (\text{map-of } D) (\text{set } lhss)$
proof –
let *?C = map-of C*
let *?D = map-of D*
let *?S = sorts-of-ssig-list C*
define *pats* **where** *pats = map ($\lambda ((f, ss), s). \text{term-and-args } f (\text{map Var } (\text{zip } [0..<\text{length } ss] ss))$) D*
have *pats: patL \in set pats \longleftrightarrow ($\exists ((f, ss), s) \in \text{set } D. \text{patL} = \text{term-and-args } f (\text{map Var } (\text{zip } [0..<\text{length } ss] ss))$)*
for *patL*
by (*force simp: pats-def split: prod.splits*)
define *P* **where** *P = map (List.maps ($\lambda \text{pat}. \text{map } (\lambda \text{lhs}. [(pat, lhs)]) \text{lhss}$)) pats*
define *V* **where** $V = \{x : \iota \text{ in } \mathcal{V}. \iota \in \text{set } (\text{sorts-of-ssig-list } C)\}$
let *?match-lhs = $\lambda t. \exists l \in \text{set } lhss. l \text{ matches } t$*
from *assms(1)*
have *b: b = decide-pat-complete rn rv 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 2
  apply unfold-locales using distC dec by auto
  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
        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 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 - ren])
  {
    fix f ss s i si
    assume mem: ((f, ss), s) ∈ set D and isi: (i, si) ∈ set (zip [0..s V → 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 :i ss in 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 → 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..

```

```

    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..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..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
    end
  end

```

```

    qed
  qed
  from L[rule-format, OF sigma 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 ∈ dom TB(?C, ?D) → (∃ ti ∈ set (t # args t). ?match-lhs
ti))
    unfolding in-dom-Basic term-and-args-def by fastforce
  finally show ?thesis unfolding strong-quasi-reducible-def by blast
  qed

```

8.1 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* ∈ dom T_B(*C, D*). ∃ *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 $\bigwedge l f ls \tau s \tau. l \in L \implies l = \text{Fun } f \text{ ls} \implies \neg f : \tau s \rightarrow \tau \text{ in } C$

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* ∈ dom T_B(*C, D*)

from *q*[*unfolded quasi-reducible-def, rule-format, OF this*]

obtain *tp* **where** *tp*: *t* ⊇ *tp* **and** *match*: ∃ *l* ∈ *L*. *l* matches *tp* **by** *auto*

```

show  $\exists l \in L. l \text{ matches } t$ 
proof (cases  $t = tp$ )
  case True
  thus ?thesis using match by auto
next
  case False
  from  $t$ [unfolded in-dom-Basic]
  obtain  $f \ ts \ ss$  where  $t = Fun \ f \ ts$  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  $subt: ti \supseteq tp$  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  $f : ss \rightarrow s$  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

```

9 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

```

9.1 FSCD paper

turn error message into runtime error

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

turn error message into runtime error

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

definition *decide-strong-qd-lhss-string* = *decide-strong-quasi-reducible* *rn-string* *rv-string*

lemmas *decide-pat-complete-lhss-string* = *decide-pat-complete-lhss*[OF - *renaming-string*,

folded decide-pat-complete-lhss-string-def]

lemmas *decide-strong-qd-lhss-string* = *decide-strong-quasi-reducible*[OF - *renaming-string*,

folded decide-strong-qd-lhss-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-fscd*:

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

lemma *decide-pat-complete-wrapper*:

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

lemma *decide-strong-quasi-reducible-wrapper*:

assumes (case *decide-strong-qd-lhss-string* *C D lhss* of *Inr b* \Rightarrow *Some b* | *Inl -*
 \Rightarrow *None*) = *Some res*
shows *strong-quasi-reducible* (*map-of C*) (*map-of D*) (*set lhss*) = *res*
using *decide-strong-qd-lhss-string*[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-fscd[of - - - True])
by eval+

lemma  $\neg$  pat-complete-lhss (map-of int-bool) (map-of even-int) (set even-lhss-int)

apply (subst decide-pat-complete-wrapper-fscd[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-fscd[of - - - True])
by eval+

lemma  $\neg$  pat-complete-lhss (map-of nat-bool) (map-of [("f",["nat","nat","nat"]),"bool"])
(set non-lin-lhss)
apply (subst decide-pat-complete-wrapper-fscd[of - - - False])
by eval+

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

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  $\neg$  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+

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+

```

```

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));
  lhsT = (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]);
  lhs = (if c3 then [lhsF] @ lhsT else lhsT @ [lhsF])
in (C, D, lhs))

```

```

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

```

definition test-problem-integer where

```

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

```

fun term-to-haskell where

```

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

```

definition createHaskellInput :: integer => integer => (integer list × integer list)

```

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

```

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

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 []

```

9.2 Journal Submission

definition *pat-complete-alg-linear* :: ((f × 's list) × 's list) ⇒ ((f × 's list) × 's list) ⇒ (f, 'v)term list ⇒ bool **where**

```

  pat-complete-alg-linear C D lhss = (

```

```

    case decide-pat-complete-linear-lhss C D lhss of Inl err ⇒ Code.abort (err (STR
"'")) (λ -. True)

```

```

    | Inr res ⇒ res)

```

definition *pat-complete-alg-new* :: ((f × 's list) × 's list) ⇒ ((f × 's list) × 's list) ⇒ (f, string)term list ⇒ bool **where**

```

  pat-complete-alg-new C D lhss = (

```

```

    case decide-pat-complete-lhss-string C D lhss of Inl err ⇒ Code.abort (err (STR
"'")) (λ -. True)

```

```

    | Inr res ⇒ res)

```

```

value (code) pat-complete-alg-linear nat-bool even-nat even-lhss

```

```

value (code) pat-complete-alg-linear int-bool even-int even-lhss-int

```

```

value (code) pat-complete-alg-fscd nat-bool [(('f',["bool","bool","bool"]),"bool")]
non-lin-lhss

```

```

value (code) pat-complete-alg-fscd nat-bool [(('f',["nat","nat","nat"]),"bool")] non-lin-lhss

```

definition *pat-complete-alg-test-linear* :: integer ⇒ integer ⇒ (integer list * integer

list)*list* ⇒ *bool* **where**

pat-complete-alg-test-linear *c n perms* = (case *test-problem-integer* *c n perms* of
(*C,D,lhss*) ⇒ *pat-complete-alg-linear* *C D lhss*)

definition *pat-complete-alg-test-new* :: *integer* ⇒ *integer* ⇒ (*integer list* * *integer list*)*list* ⇒ *bool* **where**

pat-complete-alg-test-new *c n perms* = (case *test-problem-integer* *c n perms* of
(*C,D,lhss*) ⇒ *pat-complete-alg-new* *C D lhss*)

definition *test-problem-nl1* :: *integer* ⇒ - **where**

test-problem-nl1 *n* = (let
 n' = *int-of-integer* *n*;
 s = (λ *i*. *CHR* "s" # *show* *i*);
 c = (λ *i*. ((*CHR* "c" # *show* *i*, if *i* = 0 then [] else [*s* (*i* - 1), *s* (*i* - 1)]), *s* *i*
));
 C = *map* *c* [0..*n*];
 D = [(("f", [*s* *n'*, *s* *n*]), *s* 0)];
 lhss = [*Fun* "f" [*Var* "x", *Var* "x']]
in (*C*, *D*, *lhss*)

definition *test-problem-nl2* :: *integer* ⇒ - **where**

test-problem-nl2 *n* = (case *test-problem-nl1* *n* of
(*C*, *D*, *lhss*) ⇒ (((*d*", []), "s0") # *C*, *D*, *lhss*))

definition *test-problem-nl3* :: *integer* ⇒ - **where**

test-problem-nl3 *n* = (let
 n' = *int-of-integer* (*n* + 1);
 s = (λ *i*. *CHR* "s" # *show* *i*);
 x = (λ *i*. *Var* (*CHR* "x" # *show* *i*));
 cSym = (λ *i*. *CHR* "c" # *show* *i*);
 c = (λ *i*. ((*cSym* *i*, if *i* = 0 then [] else [*s* (*i* - 1), *s* (*i* - 1)]), *s* *i*));
 C = *map* *c* [0..*n*];
 D = [(("f", *map* *s* [0..*n*]), *s* 0)];
 lhss = *map* (λ *k*. (*Fun* "f" (*map* (*x* (*k* + 1 := *Fun* (*cSym* (*k* + 1)) [*x* *k*, *x* *k*]))
[0..*n*])) [0..*n'* - 1]
in (*C*, *D*, *lhss*)

definition *test-problem-nl4* :: *integer* ⇒ - **where**

test-problem-nl4 *n* = (case *test-problem-nl3* *n* of
(*C*, *D*, *lhss*) ⇒ (((*d*", []), "s0") # *C*, *D*, *lhss*))

definition *test-pigeon-hole* :: *int* ⇒ *nat* ⇒ - **where**

test-pigeon-hole *n m* = (let
 s = "s";
 f = "f";
 x = (λ *i*. *Var* (*CHR* "x" # *show* *i*));
 y = *Var* "y";
 C = *map* (λ *i*. (((*CHR* "c" # *show* *i*), [] :: *string list*), *s*)) [0 .. *n*];
 D = [(*f*, *map* (λ -. *s*) [0..<*Suc* *m*]), *s*];

```

xs = map x [0..<Suc m];
l = (λ i j. xs [ i := y, j := y]);
lhss = List.maps (λ i. let xsi = xs [ i := y] in
  map (λ j. Fun f (xsi [ j := y ])) [Suc i ..< Suc m]) [0..<m]
in (C, D, lhss)

```

definition *test-problem-nl5* :: integer ⇒ - **where**

```
test-problem-nl5 n = test-pigeon-hole (int-of-integer n) (nat-of-integer (n + 1))
```

definition *test-problem-nl6* :: integer ⇒ - **where**

```
test-problem-nl6 n = test-pigeon-hole (int-of-integer (n + 1)) (nat-of-integer (n
+ 1))
```

definition *test-problem-nl* :: integer ⇒ - **where**

```
test-problem-nl c = (if c = 1 then test-problem-nl1
  else if c = 2 then test-problem-nl2
  else if c = 3 then test-problem-nl3
  else if c = 4 then test-problem-nl4
  else if c = 5 then test-problem-nl5
  else if c = 6 then test-problem-nl6
  else (λ -. ( [], [], [])))
```

fun *show-term* :: (string,string)term ⇒ showsl **where**

```
show-term (Var x) = (+) (String.implode x)
| show-term (Fun f []) = (+) (String.implode f)
| show-term (Fun f ts) = showsl-paren (λ s. String.implode f + STR " " +
  showsl-sep id ((+) STR " ") (map show-term ts) s)
```

definition *show-pat-complete-test-nl* :: integer ⇒ integer ⇒ String.literal **where**

```
show-pat-complete-test-nl c n = (case test-problem-nl c n of
  (C,D,lhss) ⇒ let sorts = remdups (map snd C);
    baseS = snd (hd D);
    baseC = (fst o fst o hd o filter (λ ((-,ss),s). ss = [] ∧ s = baseS)) C;
    tos = String.implode;
    s-sym = (λ ((f,ss),s). STR "(fun " + tos f + STR " " +
      (if ss = [] then tos s else STR "(-> " + showsl-sep ((+) o tos) ((+) STR " "
        ") (ss @ [s]) (STR " ')))
      + STR " '));
    rule = (λ l. let sl = show-term l (STR ""') in STR "(rule " + sl + STR " "
      + tos baseC + STR " '))
```

```
in showsl-sep (+) showsl-nl
  ([STR "(format MSTRS)"] @
  map (λ s. STR "(sort " + tos s + STR " ')) sorts @
  map s-sym C @ map s-sym D @
  map rule lhss
  )
```

) (STR ""')

value (code) show-pat-complete-test-nl 1 3

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

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

value(code) showsl (test-problem-nl1 2) (STR ""')

lemma pat-complete-alg-test-nl-new 1 2 **by** eval

value(code) showsl (test-problem-nl2 2) (STR ""')

lemma ¬ pat-complete-alg-test-nl-new 2 2 **by** eval

value(code) showsl (test-problem-nl3 2) (STR ""')

lemma pat-complete-alg-test-nl-new 3 2 **by** eval

value(code) showsl (test-problem-nl4 2) (STR ""')

lemma ¬ pat-complete-alg-test-nl-new 4 2 **by** eval

value(code) showsl (test-problem-nl5 2) (STR ""')

lemma pat-complete-alg-test-nl-new 5 2 **by** eval

value(code) showsl (test-problem-nl6 2) (STR ""')

lemma ¬ pat-complete-alg-test-nl-new 6 2 **by** eval

declare [[code drop: equal-class.equal :: bool ⇒ bool ⇒ bool]]

lemma equal-bool-code[code]:

equal-class.equal p False = (¬ p)

equal-class.equal p True = p

unfolding equal-eq **by** auto

9.3 Export Code to SML and Haskell

Connection to AGCP, which is written in SML, and SML-export of verified pattern completeness algorithms

export-code

pat-complete-alg-test-fscd

pat-complete-alg-test-linear

pat-complete-alg-test-new

pat-complete-alg-test-nl-new

pat-complete-alg-test-nl-fscd

show-pat-complete-test

```
create-agcp-input
pat-complete-alg-fscd
strong-quasi-reducible-alg
Var
in SML module-name Pat-Complete
```

Connection to FORT, which is written in Haskell, and Haskell-export of verified pattern completeness algorithms

```
export-code encodeAutomata
showAutomata
patCompleteAutomataTest
show-pat-complete-test
show-pat-complete-test-nl
pat-complete-alg-test-fscd
pat-complete-alg-test-linear
pat-complete-alg-test-new
pat-complete-alg-test-nl-new
pat-complete-alg-test-nl-fscd
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.