

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	6
5.2	Definition of Algorithm – Inference Rules	8
5.3	Soundness of the inference rules	11
6	A Multiset-Based Inference System to Decide Pattern Completeness	17
6.1	Definition of the Inference Rules	17
6.2	The evaluation cannot get stuck	19
6.3	Termination	21
6.4	Partial Correctness via Refinement	29
7	A List-Based Implementation to Decide Pattern Completeness	32
7.1	Definition of Algorithm	32
7.2	Partial Correctness of the Implementation	39
7.3	Getting the result outside the locale with assumptions	47
8	Pattern-Completeness and Related Properties	86
8.1	Connecting Pattern-Completeness, Strong Quasi-Reducibility and Quasi-Reducibility	89
9	Setup for Experiments	90
9.1	FSCD paper	90
9.2	Journal Submission	96
9.3	Export Code to SML and Haskell	100

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
  <proof>
```

```
lemma is-singleton-list2: is-singleton-list xs  $\longleftrightarrow$  ( $\exists$  x. set xs = {x})
  <proof>
```

```
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*
<proof>

lemma *dummy-fidl-solver-code[code]: dummy-fidl-solver input* = *Code.abort (STR*
"dummy fidl solver") (λ -. *dummy-fidl-solver input*)
<proof>

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*: $f y = \text{Ball } zs \ f \implies \text{Ball } (\text{insert } y \ A) \ f = \text{Ball } (zs \cup A) \ f$
<proof>

lemma *bex-insert-cong*: $f y = f z \implies \text{Bex } (\text{insert } y \ A) \ f = \text{Bex } (\text{insert } z \ A) \ f$
<proof>

lemma *not-bdd-above-natD*:
assumes $\neg \text{bdd-above } (A :: \text{nat set})$
shows $\exists x \in A. x > n$
<proof>

lemma *list-eq-nth-eq*: $xs = ys \longleftrightarrow \text{length } xs = \text{length } ys \wedge (\forall i < \text{length } ys. xs ! i = ys ! i)$
<proof>

lemma *subt-size*: $p \in \text{poss } t \implies \text{size } (t \mid\!-\! p) \leq \text{size } t$
<proof>

lemma *removeAll-remdups*: $\text{removeAll } x \ (\text{remdups } ys) = \text{remdups } (\text{removeAll } x \ ys)$
<proof>

lemma *removeAll-eq-Nil-iff*: $\text{removeAll } x \ ys = [] \longleftrightarrow (\forall y \in \text{set } ys. y = x)$
<proof>

lemma *concat-removeAll-Nil*: $\text{concat } (\text{removeAll } [] \ xss) = \text{concat } xss$
<proof>

lemma *removeAll-eq-imp-concat-eq*:
assumes $\text{removeAll } [] \ xss = \text{removeAll } [] \ xss'$
shows $\text{concat } xss = \text{concat } xss'$
<proof>

lemma *map-remdups-commute*:
assumes $\text{inj-on } f \ (\text{set } xs)$
shows $\text{map } f \ (\text{remdups } xs) = \text{remdups } (\text{map } f \ xs)$
<proof>

lemma *Uniq-False*: $\exists_{\leq 1} a. \text{False}$ *<proof>*

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

lemma *Uniq-eq-the-elem*:

assumes $UNIQ\ A$ **and** $a \in A$ **shows** $a = the\text{-}elem\ A$
 $\langle proof \rangle$

lemma *bij-betw-imp-Uniq-iff*:

assumes $bij\text{-}betw\ f\ A\ B$ **shows** $UNIQ\ A \longleftrightarrow UNIQ\ B$
 $\langle proof \rangle$

lemma *image-Uniq*: $UNIQ\ A \implies UNIQ\ (f\ 'A)$

$\langle proof \rangle$

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

$\langle proof \rangle$

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** $\mathcal{V} \equiv sort\text{-}annotated$

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

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

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

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

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

definition $tvars\text{-}match :: (f, 'v, 's)match\text{-}problem\text{-}set \Rightarrow (nat \times 's)\ set$ **where**

$tvars\text{-}match\ mp = (\bigcup (t, l) \in mp. vars\ t)$

definition $tvars\text{-}pat :: (f, 'v, 's)pat\text{-}problem\text{-}set \Rightarrow (nat \times 's)\ set$ **where**

$tvars\text{-}pat\ pp = (\bigcup mp \in pp. tvars\text{-}match\ mp)$

definition $tvars\text{-}pats :: (f, 'v, 's)pats\text{-}problem\text{-}set \Rightarrow (nat \times 's)\ set$ **where**

$tvars\text{-}pats\ P = (\bigcup pp \in P. tvars\text{-}pat\ pp)$

definition $subst\text{-}left :: (f, nat \times 's)subst \Rightarrow ((f, nat \times 's)term \times (f, 'v)term) \Rightarrow ((f, nat \times 's)term \times (f, 'v)term)$ **where**

subst-left $\tau = (\lambda(t,r). (t \cdot \tau, r))$

A definition of pattern completeness for pattern problems.

definition *match-complete-wrt* :: $(f, nat \times s, w)gsubst \Rightarrow (f, v, s)match\text{-}problem\text{-}set \Rightarrow bool$ **where**

match-complete-wrt $\sigma mp = (\exists \mu. \forall (t,l) \in mp. t \cdot \sigma = l \cdot \mu)$

lemma *match-complete-wrt-cong*:

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

and $mp: mp = mp'$

shows $match\text{-}complete\text{-}wrt\ \sigma\ mp = match\text{-}complete\text{-}wrt\ \sigma'\ mp'$

<proof>

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>

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>

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-complete $C pp \iff (\forall \sigma :_s \mathcal{V} \mid ' tvars\text{-}pat\ pp \rightarrow \mathcal{T}(C). \exists mp \in pp. match\text{-}complete\text{-}wrt\ \sigma\ mp)$

lemma *pat-completeD*:

assumes $pp: pat\text{-}complete\ C\ pp$

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

shows $\exists mp \in pp. match\text{-}complete\text{-}wrt\ \sigma\ mp$

<proof>

lemma *pat-completeI*:

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

shows $pat\text{-}complete\ C\ pp$

<proof>

lemma *tvars-pat-empty[simp]*: $tvars\text{-}pat\ \{\} = \{\}$

<proof>

lemma *pat-complete-empty[simp]*: $pat\text{-}complete\ C\ \{\} = False$

<proof>

abbreviation *pats-complete* :: $(f, s)ssig \Rightarrow (f, v, s)pats\text{-}problem\text{-}set \Rightarrow bool$ **where**

pat-complete $C P \equiv \forall pp \in P. \text{ pat-complete } C pp$

definition *finite-constr-form-mp* $:: (f, 's) \text{ sig} \Rightarrow (f, 'v, 's) \text{ match-problem-set} \Rightarrow \text{bool}$ **where**

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

definition *finite-constr-form-pat* $:: (f, 's) \text{ sig} \Rightarrow (f, 'v, 's) \text{ pat-problem-set} \Rightarrow \text{bool}$ **where**

finite-constr-form-pat $C p = (\forall mp \in p. \text{finite-constr-form-mp } C mp)$

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 $\tau c :: \text{nat} \Rightarrow \text{nat} \times 's \Rightarrow 'f \times 's \text{ list} \Rightarrow (f, \text{nat} \times 's) \text{ subst}$ **where**
 $\tau c n x = (\lambda(f, ss). \text{subst } x (\text{Fun } f (\text{map } \text{Var } (\text{indexed-from } n \text{ ss}))))$

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

fun *conflicts* $:: (f, 'v \times 's) \text{ term} \Rightarrow (f, 'v \times 's) \text{ term} \Rightarrow ('v \times 's) \text{ list option}$ **where**
 $\text{conflicts } (\text{Var } x) (\text{Var } y) = (\text{if } x = y \text{ then Some } [] \text{ else}$
 $\text{if } \text{snd } x = \text{snd } y \text{ then Some } [x, y] \text{ else None})$
 $| \text{conflicts } (\text{Var } x) (\text{Fun } -) = (\text{Some } [x])$
 $| \text{conflicts } (\text{Fun } -) (\text{Var } x) = (\text{Some } [x])$
 $| \text{conflicts } (\text{Fun } f \text{ ss}) (\text{Fun } g \text{ ts}) = (\text{if } (f, \text{length } \text{ss}) = (g, \text{length } \text{ts})$
 $\text{then map-option concat } (\text{those } (\text{map2 } \text{conflicts } \text{ss } \text{ts}))$
 $\text{else None})$

abbreviation *Conflict-Var* $s t x \equiv \text{conflicts } s t \neq \text{None} \wedge x \in \text{set } (\text{the } (\text{conflicts } s t))$

abbreviation *Conflict-Clash* $s t \equiv \text{conflicts } s t = \text{None}$

lemma *conflicts-sym: rel-option* $(\lambda xs ys. \text{set } xs = \text{set } ys) (\text{conflicts } s t) (\text{conflicts } t s)$ **(is rel-option - (?c s t) -)**
 $\langle \text{proof} \rangle$

lemma *conflicts:*

shows *Conflict-Clash* $s t \Longrightarrow$

$\exists p. p \in \text{poss } s \wedge p \in \text{poss } t \wedge$
 $(\text{is-Fun } (s \mid -p) \wedge \text{is-Fun } (t \mid -p) \wedge \text{root } (s \mid -p) \neq \text{root } (t \mid -p) \vee$
 $(\exists x y. s \mid -p = \text{Var } x \wedge t \mid -p = \text{Var } y \wedge \text{snd } x \neq \text{snd } y))$
(is ?B1 \Longrightarrow ?B2)

and *Conflict-Var* $s t x \Longrightarrow$

$\exists p. p \in \text{poss } s \wedge p \in \text{poss } t \wedge s \mid -p \neq t \mid -p \wedge$
 $(s \mid -p = \text{Var } x \vee t \mid -p = \text{Var } x)$
(is ?C1 $x \Longrightarrow$?C2 x)

and $s \neq t \implies \exists x. \text{Conflict-Clash } s t \vee \text{Conflict-Var } s t x$
and $\text{Conflict-Var } s t x \implies x \in \text{vars } s \cup \text{vars } t$
and $\text{conflicts } s t = \text{Some } [] \iff s = t \text{ (is ?A)}$
 <proof>

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

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

locale $\text{pattern-completeness-context} =$
fixes $S :: 's \text{ set}$ — set of sort-names
and $C :: ('f, 's)\text{ssig}$ — sorted signature
and $m :: \text{nat}$ — upper bound on arities of constructors
and $Cl :: 's \Rightarrow ('f \times 's \text{ list})\text{list}$ — a function to compute all constructors of given sort as list
and $\text{inf-sort} :: 's \Rightarrow \text{bool}$ — a function to indicate whether a sort is infinite
and $\text{cd-sort} :: 's \Rightarrow \text{nat}$ — a function to compute finite cardinality of a sort
and $\text{improved} :: \text{bool}$ — 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 $\text{tvvars-disj-pp} :: \text{nat set} \Rightarrow ('f, 'v, 's)\text{pat-problem-set} \Rightarrow \text{bool}$ **where**
 $\text{tvvars-disj-pp } V p = (\forall mp \in p. \forall (ti, pi) \in mp. \text{fst } ' \text{vars } ti \cap V = \{\})$

definition $\text{lvvars-disj-mp} :: 'v \text{ list} \Rightarrow ('f, 'v, 's)\text{match-problem-set} \Rightarrow \text{bool}$ **where**
 $\text{lvvars-disj-mp } ys mp = (\bigcup (\text{vars } ' \text{snd } ' mp) \cap \text{set } ys = \{\} \wedge \text{distinct } ys)$

definition $\text{inf-var-conflict} :: ('f, 'v, 's)\text{match-problem-set} \Rightarrow \text{bool}$ **where**
 $\text{inf-var-conflict } mp = (\exists s t x y.$
 $(s, \text{Var } x) \in mp \wedge (t, \text{Var } x) \in mp \wedge \text{Conflict-Var } s t y \wedge \text{inf-sort } (\text{snd } y))$

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

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

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

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

(**infix** $\langle \rightarrow_s \rangle$ 50) **where**

$mp\text{-decompose}: \text{length } ts = \text{length } ls \Longrightarrow \text{insert } (\text{Fun } f \ ts, \text{Fun } f \ ls) \ mp \rightarrow_s \text{ set } (\text{zip } ts \ ls) \cup mp$

| $mp\text{-match}: x \notin \bigcup (\text{vars } ' \text{snd } ' \ mp) \Longrightarrow \text{insert } (t, \text{Var } x) \ mp \rightarrow_s mp$

| $mp\text{-identity}: mp \rightarrow_s mp$

| $mp\text{-decompose}': mp \cup mp' \rightarrow_s (\bigcup (t, l) \in mp. \text{set } (\text{zip } (\text{args } t) (\text{map } \text{Var } ys))) \cup mp'$

if $\bigwedge t \ l. (t, l) \in mp \Longrightarrow l = \text{Var } y \wedge \text{root } t = \text{Some } (f, n)$

$\bigwedge t \ l. (t, l) \in mp' \Longrightarrow y \notin \text{vars } l$

$\text{tvars-disj-mp } ys \ (mp \cup mp') \ \text{length } ys = n$

improved

inductive $mp\text{-fail} :: ('f, 'v, 's)\text{match-problem-set} \Rightarrow \text{bool}$ **where**

$mp\text{-clash}: (f, \text{length } ts) \neq (g, \text{length } ls) \Longrightarrow mp\text{-fail } (\text{insert } (\text{Fun } f \ ts, \text{Fun } g \ ls) \ mp)$

| $mp\text{-clash}': \text{Conflict-Clash } s \ t \Longrightarrow mp\text{-fail } (\{(s, \text{Var } x), (t, \text{Var } x)\} \cup mp)$

| $mp\text{-clash-sort}: \mathcal{T}(C, \mathcal{V}) \ s \neq \mathcal{T}(C, \mathcal{V}) \ t \Longrightarrow mp\text{-fail } (\{(s, \text{Var } x), (t, \text{Var } x)\} \cup mp)$

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

(**infix** $\langle \Rightarrow_s \rangle$ 50) **where**

$pp\text{-simp-mp}: mp \rightarrow_s mp' \Longrightarrow \text{insert } mp \ pp \Rightarrow_s \{\text{insert } mp' \ pp\}$

| $pp\text{-remove-mp}: mp\text{-fail } mp \Longrightarrow \text{insert } mp \ pp \Rightarrow_s \{pp\}$

| $pp\text{-success}: \text{insert } \{\} \ pp \Rightarrow_s \{\}$

| $pp\text{-inf-var-conflict}: pp \cup pp' \Rightarrow_s \{pp'\}$

if $\text{Ball } pp \ \text{inf-var-conflict}$

$\text{finite } pp$

$\text{Ball } (\text{tvars-pat } pp') \ (\lambda x. \neg \text{inf-sort } (\text{snd } x))$

$\neg \text{improved} \Longrightarrow pp' = \{\}$

| $pp\text{-instantiate}: \text{tvars-disj-pp } \{n \ .. < n+m\} \ pp \Longrightarrow x \in \text{tvars-pat } pp \Longrightarrow$

$pp \Rightarrow_s \{\text{subst-pat-problem-set } \tau \ pp \mid \tau \in \tau s \ n \ x\}$

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

Example-program: $f(x, x) = \dots, f(s(x), y) = \dots, f(x, s(y)) = \dots$ cover all cases of natural numbers, i.e., $f(x_1, x_2)$, but if one would immediately delete the matching problem of the first lhs because of the resulting inf-var-conflict in $(x_1, x), (x_2, x)$ then it is no longer complete.

inductive $P\text{-step-set} :: ('f, 'v, 's)\text{pats-problem-set} \Rightarrow ('f, 'v, 's)\text{pats-problem-set} \Rightarrow \text{bool}$

(**infix** $\langle \Rightarrow_s \rangle$ 50) **where**

$P\text{-fail}: \text{insert } \{\} \ P \Rightarrow_s \text{bottom}$

| $P\text{-simp}: pp \Rightarrow_s P' \Longrightarrow \text{insert } pp \ P \Rightarrow_s P' \cup P$

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

lemma $wf\text{-}match\text{-}iff$: $wf\text{-}match\ mp \longleftrightarrow (\forall (x, \iota) \in tvars\text{-}match\ mp. \iota \in S)$
 $\langle proof \rangle$

lemma $tvars\text{-}match\text{-}subst$: $tvars\text{-}match\ (subst\text{-}match\text{-}problem\text{-}set\ \sigma\ mp) = (\bigcup (t, l) \in mp. vars\ (t \cdot \sigma))$
 $\langle proof \rangle$

lemma $wf\text{-}match\text{-}subst$:

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 $wf\text{-}match\ (subst\text{-}match\text{-}problem\text{-}set\ \sigma\ mp)$
 $\langle proof \rangle$

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

lemma $wf\text{-}pat\text{-}subst$:

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 $wf\text{-}pat\ (subst\text{-}pat\text{-}problem\text{-}set\ \sigma\ pp)$
 $\langle proof \rangle$

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

lemma $wf\text{-}pat\text{-}iff$: $wf\text{-}pat\ pp \longleftrightarrow (\forall (x, \iota) \in tvars\text{-}pat\ pp. \iota \in S)$
 $\langle proof \rangle$

The reduction of match problems preserves completeness.

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

lemma $mp\text{-}fail\text{-}pcorrect1$:

assumes $mp\text{-}fail\ mp\ \sigma :_s\ sort\text{-}annotated \mid 'tvars\text{-}match\ mp \rightarrow \mathcal{T}(C, X)$
shows $\neg match\text{-}complete\text{-}wrt\ \sigma\ mp$
 $\langle proof \rangle$

lemma $mp\text{-}fail\text{-}pcorrect$:

assumes $f: mp\text{-}fail\ mp$ **and** $s: \sigma :_s \{x : \iota \text{ in } \mathcal{V}. \iota \in S\} \rightarrow \mathcal{T}(C)$ **and** $wf: wf\text{-}match\ mp$
shows $\neg match\text{-}complete\text{-}wrt\ \sigma\ mp$
 $\langle proof \rangle$

abbreviation SS where $SS \equiv (UNIV :: nat\ set) \times S$

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
 $C\ m\ Cl\ inf\text{-}sort\ cd\text{-}sort$

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

begin

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

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

lemma *C-nth-S*: $f : ss \rightarrow s\ in\ C \implies i < length\ ss \implies ss!i \in S$
<proof>

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

Preservation of well-formedness

lemma *mp-step-wf*: $mp \rightarrow_s mp' \implies wf\text{-}match\ mp \implies wf\text{-}match\ mp'$
<proof>

lemma *pp-step-wf*: $pp \Rightarrow_s P' \implies wf\text{-}pat\ pp \implies pp' \in P' \implies wf\text{-}pat\ pp'$
<proof>

theorem *P-step-set-wf*: $P \Rightarrow_s P' \implies wf\text{-}pats\ P \implies wf\text{-}pats\ P'$
<proof>

Soundness requires some preparations

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

lemma $\sigma g: \sigma g :_s \{x : \iota \text{ in sort-annotated. } \iota \in S\} \rightarrow \mathcal{T}(C, \emptyset)$
 $\langle \text{proof} \rangle$

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)
 $\langle \text{proof} \rangle$

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)
 $\langle \text{proof} \rangle$

lemma *inf-var-conflictD*: **assumes** *inf-var-conflict mp*

shows $\exists p \ s \ t \ x \ y.$
 $(s, \text{Var } x) \in mp \wedge (t, \text{Var } x) \in mp \wedge s \mid -p = \text{Var } y \wedge s \mid -p \neq t \mid -p \wedge$
 $p \in \text{poss } s \wedge p \in \text{poss } t \wedge \text{inf-sort } (\text{snd } y)$
 $\langle \text{proof} \rangle$

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

lemma $\sigma g': \sigma g' :_s \mathcal{V} \mid 'SS \rightarrow \mathcal{T}(C)$
 $\langle \text{proof} \rangle$

lemma *typed-imp-S*: $t : \iota \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS) \implies \iota \in S$
 $\langle \text{proof} \rangle$

lemma *typed-S-eq*: **assumes** $t : t : \tau \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS)$

and $t': t : \iota \text{ in } \mathcal{T}(C, \mathcal{V})$
shows $\tau = \iota$
 $\langle \text{proof} \rangle$

lemma *finite-arg-sort*:

assumes *finite-sort C ι*
and $f : f : \sigma s \rightarrow \iota \text{ in } C$
and $\sigma : \sigma \in \text{set } \sigma s$
shows *finite-sort C σ*
 $\langle \text{proof} \rangle$

lemma *finite-arg-sorts*:

assumes *finite-sort* $C \iota$
and *Fun fts* : ι in $\mathcal{T}(C, V)$
and $t \in \text{sets}$
shows $\exists \iota. t : \iota$ in $\mathcal{T}(C, V) \wedge \text{finite-sort } C \iota$
 $\langle \text{proof} \rangle$

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

lemma *pp-step-pcorrect*:

$pp \Rightarrow_s P' \implies \text{wf-pat } pp \implies \text{pat-complete } C \text{ } pp = \text{pats-complete } C \text{ } P'$
 $\langle \text{proof} \rangle$

theorem *P-step-set-pcorrect*:

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

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**

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

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 $\text{match-of-var-form } (\text{var-form-of-match } mp) = mp$

$\langle \text{proof} \rangle$

lemma *tvars-match-var-form*:

assumes *var-form-match* mp

shows $\text{tvars-match } mp = \{v. \exists x. (Var \ v, Var \ x) \in mp\}$

$\langle \text{proof} \rangle$

lemma *pat-of-var-form-pat*:

assumes *var-form-pat* pp

shows $\text{pat-of-var-form } (\text{var-form-of-pat } pp) = pp$

$\langle \text{proof} \rangle$

lemma *tvars-pat-var-form*: $\text{tvars-pat } (\text{pat-of-var-form } ff) = \text{tvars-var-form-pat } ff$

<proof>

lemma *tvars-var-form-pat*:
assumes *var-form-pat pp*
shows *tvars-var-form-pat (var-form-of-pat pp) = tvars-pat pp*
<proof>

lemma *pat-complete-var-form*:
pat-complete C (pat-of-var-form ff) \longleftrightarrow
($\forall \sigma :_s \mathcal{V} \mid 'tvars\text{-var-form-pat } ff \rightarrow \mathcal{T}(C). \exists f \in ff. \exists \mu. \forall x. \forall y \in f x. \sigma y = \mu$
x)
<proof>

lemma *pat-complete-var-form-set*:
pat-complete C (pat-of-var-form ff) \longleftrightarrow
($\forall \sigma :_s \mathcal{V} \mid 'tvars\text{-var-form-pat } ff \rightarrow \mathcal{T}(C). \exists f \in ff. \exists \mu. \forall x. \sigma 'f x \subseteq \{\mu x\}$
<proof>

lemma *pat-complete-var-form-Uniq*:
pat-complete C (pat-of-var-form ff) \longleftrightarrow
($\forall \sigma :_s \mathcal{V} \mid 'tvars\text{-var-form-pat } ff \rightarrow \mathcal{T}(C). \exists f \in ff. \forall x. \text{UNIQ } (\sigma 'f x)$
<proof>

lemma *ex-var-form-pat*: *($\exists f \in \text{var-form-of-pat } pp. P f$) \longleftrightarrow ($\exists mp \in pp. P (\text{var-form-of-match } mp)$)*
<proof>

lemma *pat-complete-var-form-nat*:
assumes *fin: $\forall (x,l) \in \text{tvars-var-form-pat } ff. \text{finite-sort } C \iota$*
and *uniq: $\forall f \in ff. \forall x::'v. \text{UNIQ } (\text{snd } 'f x)$*
shows *pat-complete C (pat-of-var-form ff) \longleftrightarrow*
($\forall \alpha. (\forall v \in \text{tvars-var-form-pat } ff. \alpha v < \text{card-of-sort } C (\text{snd } v)) \longrightarrow$
($\exists f \in ff. \forall x. \text{UNIQ } (\alpha 'f x)$)
(is ?l \longleftrightarrow ($\forall \alpha. ?s \alpha \longrightarrow ?r \alpha$))
<proof>

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

lemma *UNIQ-subst-pairI*: **assumes** $\bigwedge s t. s \in A \implies t \in A \implies s \cdot \sigma = t \cdot \sigma$
shows *UNIQ-subst σA <proof>*

lemma *UNIQ-subst-trivial[simp]*: *UNIQ-subst $\sigma \{t\}$ UNIQ-subst $\sigma \{\}$*

<proof>

lemma *UNIQ-subst-pairD*: **assumes** *UNIQ-subst* σ *A*
shows $s \in A \implies t \in A \implies s \cdot \sigma = t \cdot \sigma$
<proof>

lemma *UNIQ-mono*: **assumes** $A \subseteq B$
shows *UNIQ* $B \implies$ *UNIQ* A *<proof>*

lemma *UNIQ-subst-mono*: **assumes** $A \subseteq B$
shows *UNIQ-subst* σ $B \implies$ *UNIQ-subst* σ A
<proof>

lemma *UNIQ-subst-alt-def*: *UNIQ-subst* σ $A = (\forall s t. s \in A \longrightarrow t \in A \longrightarrow s \cdot \sigma = t \cdot \sigma)$
<proof>

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

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

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

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

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

lemma *tvars-spat-cong*: **assumes** $\bigwedge x. x \in$ *tvars-spat* $spp \implies \sigma x = \delta x$
and $mp \in spp$
shows *simple-match-complete-wrt* σ $mp = simple-match-complete-wrt \delta$ mp
<proof>

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 $mp = (\forall eqc \in mp. eqc \neq \{\}) \wedge (\exists \iota. finite-sort C \iota)$

$\wedge (\forall t \in \text{eqc. } t : \iota \text{ in } \mathcal{T}(C, \mathcal{V} \mid 'SS)))$

definition *finite-constructor-form-pat* $p = \text{Ball } p \text{ finite-constructor-form-mp}$

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

definition *fcf-solver* **where**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

inductive $\text{mp-step-mset} :: ('f, 'v, 's)\text{match-problem-mset} \Rightarrow ('f, 'v, 's)\text{match-problem-mset}$
 $\Rightarrow \text{bool}$ (**infix** $\langle \rightarrow_m \rangle$ 50) **where**
 $\text{match-decompose}: (f, \text{length } ts) = (g, \text{length } ls)$
 $\implies \text{add-mset } (\text{Fun } f \ ts, \text{Fun } g \ ls) \ \text{mp} \rightarrow_m \ \text{mp} + \text{mset } (\text{zip } ts \ ls)$
| $\text{match-match}: x \notin \bigcup (\text{vars } \text{'snd } \text{'set-mset } \text{mp})$
 $\implies \text{add-mset } (t, \text{Var } x) \ \text{mp} \rightarrow_m \ \text{mp}$
| $\text{match-duplicate}: \text{add-mset pair } (\text{add-mset pair } \text{mp}) \rightarrow_m \text{add-mset pair } \text{mp}$
| $\text{match-decompose}' : \text{mp} + \text{mp}' \rightarrow_m (\sum (t, l) \in \# \ \text{mp}. \text{mset } (\text{zip } (\text{args } t) (\text{map } \text{Var } ys))) + \text{mp}'$
if $\bigwedge t \ l. (t, l) \in \# \ \text{mp} \implies l = \text{Var } y \wedge \text{root } t = \text{Some } (f, n)$
 $\bigwedge t \ l. (t, l) \in \# \ \text{mp}' \implies y \notin \text{vars } l$
 $\text{lvars-disj-mp } ys \ (\text{mp-mset } (\text{mp} + \text{mp}')) \ \text{length } ys = n$
 $\text{size } \text{mp} \geq 2$
improved

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

inductive $\text{pp-step-mset} :: ('f, 'v, 's)\text{pat-problem-mset} \Rightarrow ('f, 'v, 's)\text{pats-problem-mset}$
 $\Rightarrow \text{bool}$

(**infix** $\langle \Rightarrow_m \rangle$ 50) **where**
pat-remove-pp: $add\text{-}mset \ \{\#\} \ pp \Rightarrow_m \ \{\#\}$
| *pat-simp-mp*: $mp\text{-}step\text{-}mset \ mp \ mp' \Longrightarrow add\text{-}mset \ mp \ pp \Rightarrow_m \ \{\#\} \ (add\text{-}mset \ mp' \ pp) \ \#\}$
| *pat-remove-mp*: $match\text{-}fail \ mp \Longrightarrow add\text{-}mset \ mp \ pp \Rightarrow_m \ \{\#\} \ pp \ \#\}$
| *pat-instantiate*: $tvars\text{-}disj\text{-}pp \ \{n \ ..< \ n+m\} \ (pat\text{-}mset \ (add\text{-}mset \ mp \ pp)) \Longrightarrow$
 $(Var \ x, \ l) \in mp\text{-}mset \ mp \wedge is\text{-}Fun \ l \vee$
 $\neg improved \wedge (s, Var \ y) \in mp\text{-}mset \ mp \wedge (t, Var \ y) \in mp\text{-}mset \ mp \wedge Conflict\text{-}Var$
 $s \ t \ x \wedge \neg inf\text{-}sort \ (snd \ x)$
 \Longrightarrow
 $add\text{-}mset \ mp \ pp \Rightarrow_m \ mset \ (map \ (\lambda \ \tau. \ subst\text{-}pat\text{-}problem\text{-}mset \ \tau \ (add\text{-}mset \ mp \ pp)) \ (\tau s\text{-}list \ n \ x))$
| *pat-inf-var-conflict*: $Ball \ (pat\text{-}mset \ pp) \ inf\text{-}var\text{-}conflict \Longrightarrow pp \neq \ \{\#\}$
 $\Longrightarrow Ball \ (tvars\text{-}pat \ (pat\text{-}mset \ pp')) \ (\lambda \ x. \ \neg \ inf\text{-}sort \ (snd \ x)) \Longrightarrow$
 $(\neg improved \Longrightarrow pp' = \ \{\#\})$
 $\Longrightarrow pp + pp' \Rightarrow_m \ \{\#\} \ pp' \ \#\}$

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

inductive $P\text{-}step\text{-}mset :: ('f, 'v, 's)pat\text{-}problem\text{-}mset \Rightarrow ('f, 'v, 's)pat\text{-}problem\text{-}mset$
 $\Rightarrow bool$

(**infix** $\langle \Rightarrow_m \rangle$ 50) **where**
P-failure: $add\text{-}mset \ \{\#\} \ P \neq bottom\text{-}mset \Longrightarrow add\text{-}mset \ \{\#\} \ P \Rightarrow_m bottom\text{-}mset$
| *P-simp-pp*: $pp \Rightarrow_m pp' \Longrightarrow add\text{-}mset \ pp \ P \Rightarrow_m pp' + P$

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

definition $P\text{-}step :: (('f, 'v, 's)pat\text{-}problem\text{-}mset \times ('f, 'v, 's)pat\text{-}problem\text{-}mset) \ set$
 $(\langle \Rightarrow \rangle)$ **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*:
 $\exists n. \ tvar\text{-}disj\text{-}pp \ \{n..<n+m\} \ (pat\text{-}mset \ p)$
 $\langle proof \rangle$

lemma *mp-mset-in-pat-mset*: $mp \in \# pp \Longrightarrow mp\text{-}mset \ mp \in pat\text{-}mset \ pp$
 $\langle proof \rangle$

lemma *mp-step-mset-cong*:
assumes $(\rightarrow_m)^{**} \ mp \ mp'$

shows $(\text{add-mset } (\text{add-mset } mp \ p) \ P, \text{add-mset } (\text{add-mset } mp' \ p) \ P) \in \Rightarrow^*$
 $\langle \text{proof} \rangle$

lemma *mp-step-mset-vars*: **assumes** $mp \rightarrow_m mp'$
shows $\text{tvars-match } (mp\text{-mset } mp) \supseteq \text{tvars-match } (mp\text{-mset } mp')$
 $\langle \text{proof} \rangle$

lemma *mp-step-mset-steps-vars*: **assumes** $(\rightarrow_m)^{**} \ mp \ mp'$
shows $\text{tvars-match } (mp\text{-mset } mp) \supseteq \text{tvars-match } (mp\text{-mset } mp')$
 $\langle \text{proof} \rangle$

end

lemma *count-le-size*: $\text{count } A \ x \leq \text{size } A$
 $\langle \text{proof} \rangle$

lemma *Max-le-MaxI*: **assumes** $\text{finite } A \ A \neq \{\} \ \text{finite } B$
 $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$
shows $\text{Max } A \leq \text{Max } B$
 $\langle \text{proof} \rangle$

lemma *steps-bound*: **assumes** $\bigwedge x \ y. (x,y) \in r \implies f \ x > f \ y$
and $(x,y) \in r \widetilde{n}$
shows $f \ x \geq f \ y + n$
 $\langle \text{proof} \rangle$

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 $\text{inf: improved} \implies \text{infinite } (\text{UNIV} :: 'v \ \text{set})$ **and** $\text{wf: wf-pat } (\text{pat-mset } p)$
shows $p = \{\#\} \vee (\exists \ ps. p \Rightarrow_m ps) \vee (\text{improved} \wedge \text{finite-constr-form-pat } C \ (\text{pat-mset } p))$
 $\langle \text{proof} \rangle$

context

assumes $\text{non-improved: } \neg \ \text{improved}$
begin

lemma *pat-empty-or-trans*: $\text{wf-pat } (\text{pat-mset } p) \implies p = \{\#\} \vee (\exists \ ps. p \Rightarrow_m ps)$
 $\langle \text{proof} \rangle$

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

theorem *P-step-NF*:

assumes $\text{wf: wf-pats } (\text{pats-mset } P)$ **and** $\text{NF: } P \in \text{NF} \implies$
shows $P \in \{\{\#\}, \text{bottom-mset}\}$

<proof>
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 $p = \{\#\} \vee (\exists ps. p \Rightarrow_m ps) \vee \text{finite-constr-form-pat } C \text{ (pat-mset } p)$
<proof>

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

theorem *P-step-NF-fvf*:
assumes *wf*: *wf-pats* (*pats-mset P*)
and *NF*: ($P :: ('f,'v,'s) \text{ pats-problem-mset} \in NF \Rightarrow$
and $p: p \in \# P$
shows *finite-constr-form-pat* *C* (*pat-mset p*)
<proof>

lemma *pp-step-mset-empty-cong*: **assumes** *improved*
shows $p \Rightarrow_m P \Rightarrow p + \text{replicate-mset } n \ \{\#\} \Rightarrow_m \text{image-mset } (\lambda p'. p' +$
 $\text{replicate-mset } n \ \{\#\}) P$
<proof>

theorem *nd-step-NF-fvf*: **fixes** *p* :: ('f,'v,'s) *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>
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 *length ts = length ls* then
sum-list (*map2 fun-diff ls ts*) else 0)
| *fun-diff* *l t* = 0

lemma *fun-diff-Var[simp]*: *fun-diff* (*Var x*) *t* = 0
<proof>

lemma *add-many-mult*: $(\bigwedge y. y \in\# N \implies (y,x) \in R) \implies (N + M, \text{add-mset } x M) \in \text{mult } R$
 ⟨proof⟩

lemma *fun-diff-num-funs*: $\text{fun-diff } l t \leq \text{num-funs } l$
 ⟨proof⟩

lemma *fun-diff-subst*: $\text{fun-diff } l (t \cdot \sigma) \leq \text{fun-diff } l t$
 ⟨proof⟩

lemma *fun-diff-num-funs-lt*: **assumes** $t': t' = \text{Fun } c \text{ cs}$
and $\text{is-Fun } l$
shows $\text{fun-diff } l t' < \text{num-funs } l$
 ⟨proof⟩

lemma *sum-union-le-nat*: $\text{sum } (f :: 'a \Rightarrow \text{nat}) (A \cup B) \leq \text{sum } f A + \text{sum } f B$
 ⟨proof⟩

lemma *sum-le-sum-list-nat*: $\text{sum } f (\text{set } xs) \leq (\text{sum-list } (\text{map } f xs) :: \text{nat})$
 ⟨proof⟩

lemma *bdd-above-has-Maximum-nat*: $\text{bdd-above } (A :: \text{nat set}) \implies A \neq \{\} \implies \text{has-Maximum } A$
 ⟨proof⟩

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

lemma *vars-term-syms-term*: $x \in \text{vars-term } t \iff \text{Inl } x \in\# \text{syms-term } t$
 ⟨proof⟩

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

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 ?!**
 $t = ?r t$)
 ⟨proof⟩

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]*: $\text{num-syms } t > 0$

<proof>

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

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>

lemma *num-syms-Fun[simp]*: $\text{num-syms } (\text{Fun } f \ ts) = \text{Suc } (\text{sum-list } (\text{map } \text{num-syms } ts))$
<proof>

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*: $\text{sum-ms } f \ (\text{image-mset } g \ ms) = \text{sum-ms } (f \ o \ g) \ ms$
<proof>

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

lemma *τs -list*: $\text{set } (\tau s\text{-list } n \ x) = \tau s \ n \ x$
<proof>

lemma *num-syms- τc* : $\text{num-syms } (t \cdot \tau c \ n \ x \ (f, \sigma s)) = \text{num-syms } t + \text{count } (\text{syms-term } t) \ (\text{Inl } x) * \text{length } \sigma s$
<proof>

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>

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

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

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

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

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

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

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

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

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

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

definition max-dupl-mp **where**

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

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

lemma $\text{num-funs-le-num-syms}$: $\text{num-funs } t \leq \text{num-syms } t$
 $\langle \text{proof} \rangle$

lemma $\text{fun-diff-le-num-syms}$: $\text{fun-diff } l \ t \leq \text{num-syms } l$
 $\langle \text{proof} \rangle$

lemma $\text{meas-diff-mp-le-num-lsyms-mp}$: $\text{meas-diff-mp } mp \leq \text{num-lsyms-mp } mp$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma $\text{tsyms-mp-num-tsyms}$: $\text{num-tsyms-mp } mp = \text{size } (\text{tsyms-mp } mp)$
 $\langle \text{proof} \rangle$

lemma $\text{meas-dupl-le-num-syms-pat}$: $\text{meas-dupl } p \leq \text{num-syms-pat } p$
 $\langle \text{proof} \rangle$

lemma *meas-diff-le-num-syms-pat*: $\text{meas-diff } p \leq \text{num-syms-pat } p$
 ⟨proof⟩

lemma *tsyms-mp-subset-num-tsyzms*: $\text{tsyms-mp } mp \subseteq\# \text{tsyms-mp } mp' \implies \text{num-tsyzms-mp } mp < \text{num-tsyzms-mp } mp'$
 ⟨proof⟩

lemma *tsyms-mp-mono*: **assumes** $mp \subseteq\# mp'$ **shows** $\text{tsyms-mp } mp \subseteq\# \text{tsyms-mp } mp'$
 ⟨proof⟩

lemma *tsyms-mp-strict-mono*: **assumes** $mp \subseteq\# mp'$ **shows** $\text{tsyms-mp } mp \subseteq\# \text{tsyms-mp } mp'$
 ⟨proof⟩

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*: $\text{measure-pat-poly } c \ p \leq (c + \text{num-syms-pat } p) * (\text{num-syms-pat } p * m + 2)$
 ⟨proof⟩

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⟩

lemma *measure-pat-poly-meas-diff*: **assumes** $\text{meas-diff } p < \text{meas-diff } p'$
and $\text{meas-dupl } p \leq \text{meas-dupl } p'$
and $\text{meas-tsyzms } p \leq \text{meas-tsyzms } p' + \text{meas-dupl } p' * m$
shows $\text{measure-pat-poly } c \ p < \text{measure-pat-poly } c \ p'$
 ⟨proof⟩

lemma *measure-pat-poly-num-syms*: **assumes** $\text{meas-diff } p \leq \text{meas-diff } p'$
and $\text{meas-dupl } p \leq \text{meas-dupl } p'$
and $\text{meas-tsyzms } p < \text{meas-tsyzms } p'$
shows $\text{measure-pat-poly } c \ p < \text{measure-pat-poly } c \ p'$
 ⟨proof⟩

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

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

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

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

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

abbreviation *gt-rel-pats* (**infix** $\langle \cdot \rangle_{mul}$ 50) **where**
P \succ_{mul} P' \equiv (P', P) $\in \langle \cdot \rangle_{mul}$

lemma *wf-rel-pat*: wf \prec
 $\langle proof \rangle$

lemma *wf-rel-pat'*: wf *rel-pat'*
 $\langle proof \rangle$

lemma *wf-rel-pats*: wf \prec_{mul}
 $\langle proof \rangle$

lemma *rel-pat-sub-rel-pat'*: *rel-pat* \subseteq *rel-pat'*
 $\langle proof \rangle$

lemma *tvars-match-fin*:
finite (tvars-match (mp-mset mp))
 $\langle proof \rangle$

lemmas *meas-def = meas-finvars-def meas-diff-def meas-tsymbols-def meas-setsize-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')
 $\langle proof \rangle$

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'
 $\langle proof \rangle$

lemma *rel-mp-sub*: $\{\#$ add-mset p mp $\#\}$ \succ $\{\#$ mp $\#\}$
 $\langle proof \rangle$

lemma *mp-step-tsyms-mp-psubset*:

fixes $mp :: ('f, 'v, 's) \text{ match-problem-mset}$
assumes $mp \rightarrow_m mp'$
shows $\text{tsyms-mp } mp' \subseteq\# \text{tsyms-mp } mp$
 $\langle \text{proof} \rangle$

lemma $\text{tvars-match-tsyms-mp}$: $\text{tvars-match } (mp\text{-mset } mp) = \{ x. \text{Inl } x \in\# \text{tsyms-mp } mp \}$
 $\langle \text{proof} \rangle$

lemma max-dupl-mp-mono : **assumes** $\text{tsyms-mp } mp \subseteq\# \text{tsyms-mp } mp'$
shows $\text{max-dupl-mp } mp \leq \text{max-dupl-mp } mp'$
 $\langle \text{proof} \rangle$

lemma $\text{mp-step-mset-meas-max-dupl}$: **assumes** $mp \rightarrow_m mp'$
shows $\text{max-dupl-mp } mp' \leq \text{max-dupl-mp } mp$
 $\langle \text{proof} \rangle$

lemma $\text{mp-step-mset-meas-finvars}$: **assumes** $mp \rightarrow_m mp'$
shows $\text{meas-finvars-mp } mp' \leq \text{meas-finvars-mp } mp$
 $\langle \text{proof} \rangle$

lemma $\text{mp-step-mset-num-tsyms-mp}$: **assumes** $mp \rightarrow_m mp'$
shows $\text{num-tsyms-mp } mp' < \text{num-tsyms-mp } mp$
 $\langle \text{proof} \rangle$

lemma $\text{mp-step-mset-meas-diff-mp}$:
fixes $mp :: ('f, 'v, 's) \text{ match-problem-mset}$
assumes $mp \rightarrow_m mp'$
shows $\text{meas-diff-mp } mp' \leq \text{meas-diff-mp } mp$
 $\langle \text{proof} \rangle$

lemma $\text{rel-mp-mp-step-mset}$:
fixes $mp :: ('f, 'v, 's) \text{ match-problem-mset}$
assumes $\text{step: } mp \rightarrow_m mp'$
shows $\{\#mp\# \} > \{\#mp'\#\}$
 $\langle \text{proof} \rangle$

lemma $\text{mp-step-measure-pat-poly}$:
fixes $mp :: ('f, 'v, 's) \text{ match-problem-mset}$
assumes $\text{step: } mp \rightarrow_m mp'$
shows $\text{measure-pat-poly } c (\text{add-mset } mp \ p) > \text{measure-pat-poly } c (\text{add-mset } mp' \ p)$
 $\langle \text{proof} \rangle$

lemma $\text{meas-diff-subst-le}$: $\text{meas-diff } (\text{subst-pat-problem-mset } \tau \ p) \leq \text{meas-diff } p$
 $\langle \text{proof} \rangle$

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

lemma *meas-sub-rel-pat*: **assumes** *sub*: $p' \subset_{\#} p$
shows $(p', p) \in \text{rel-pat}'$
⟨*proof*⟩

lemma *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*⟩

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 $\text{size } t \leq \text{max-size } s$
⟨*proof*⟩

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*⟩

lemma *add-mset-rel-pat*: **assumes** *sub*: $mp \neq \{\#\}$
shows $\text{add-mset } mp \succ p$
⟨*proof*⟩

lemma *add-mset-measure-pat-poly*: **assumes** *sub*: $mp \neq \{\#\}$
shows $\text{measure-pat-poly } c (\text{add-mset } mp \ p) > \text{measure-pat-poly } c \ p$
⟨*proof*⟩

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

lemma *meas-tsymbols-inst*: **fixes** $p :: ('f, 'v, 's) \text{ pat-problem-mset}$
assumes $\tau \in \tau s \ n \ x$
shows $\text{meas-tsymbols } (\text{subst-pat-problem-mset } \tau \ p) \leq \text{meas-tsymbols } p + \text{meas-dupl } p * m$
⟨*proof*⟩

lemma *pp-step-le-size*: **assumes** $p \Rightarrow_m ps$ **and** $p' \in_{\#} ps$
shows $\text{size } p' \leq \text{size } p$

$\langle proof \rangle$

lemma *decrease-pp-step-mset*:

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

assumes $p \Rightarrow_m ps$

and $p' \in \# ps$

shows $p \succ p' \text{ improved} \implies \text{measure-pat-poly } c \ p > \text{measure-pat-poly } c \ p'$

$\langle proof \rangle$

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

lemma *rel-P-trans*:

assumes $P \Rightarrow_m P'$

shows $P \succ_{mul} P'$

$\langle proof \rangle$

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

lemma *nd-step-le-size*: **assumes** $(p, q) \in \Rightarrow_{nd}$

shows $\text{size } q \leq \text{size } p$

$\langle proof \rangle$

lemma *nd-steps-le-size*: **assumes** $(p, q) \in \Rightarrow_{nd}^*$

shows $\text{size } q \leq \text{size } p$

$\langle proof \rangle$

lemma *nd-step-decrease*: **assumes** $(p, q) \in \Rightarrow_{nd}$

shows $p \succ q$

improved $\implies \text{measure-pat-poly } c \ p > \text{measure-pat-poly } c \ q$

$\langle proof \rangle$

lemma *nd-steps-bound*: **assumes** *improved*

and $(p, q) \in \Rightarrow_{nd} \tilde{n}$

shows $n \leq \text{measure-pat-poly } c \ p$ (**is** ?A)

$\text{measure-pat-poly } c \ q + n \leq \text{measure-pat-poly } c \ p$ (**is** ?B)

$\langle proof \rangle$

theorem *SN-nd-pstep*: $SN \Rightarrow_{nd}$

$\langle proof \rangle$

theorem *SN-P-step*: $SN \Rightarrow$

$\langle proof \rangle$

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*: $mp1 \rightarrow_s mp2 \implies mp1 = mp1' \implies mp2 = mp2' \implies mp1' \rightarrow_s mp2'$ $\langle proof \rangle$

lemma *mp-step-mset-mp-trans*: $mp \rightarrow_m mp' \implies mp\text{-mset } mp \rightarrow_s mp\text{-mset } mp'$
 ⟨proof⟩

lemma *mp-fail-cong*: $mp\text{-fail } mp \implies mp = mp' \implies mp\text{-fail } mp'$ ⟨proof⟩

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

lemma *pp-step-set-cong*: $P \Rightarrow_s Q \implies P = P' \implies Q = Q' \implies P' \Rightarrow_s Q'$ ⟨proof⟩

lemma *p-step-mset-imp-set*: **assumes** $p \Rightarrow_m Q$
shows $pat\text{-mset } p \Rightarrow_s pats\text{-mset } Q$
 ⟨proof⟩

lemma *pp-step-mset-pcorrect*: $p \Rightarrow_m P' \implies wf\text{-pat } (pat\text{-mset } p) \implies$
 $pat\text{-complete } C (pat\text{-mset } p) = pats\text{-complete } C (pats\text{-mset } P')$
 ⟨proof⟩

lemma *P-step-mset-imp-set*: **assumes** $P \Rightarrow_m Q$
shows $pats\text{-mset } P \Rightarrow_s pats\text{-mset } Q$
 ⟨proof⟩

lemma *nd-step-mset-pcorrect*: **assumes** $p \notin NF \Rightarrow_{nd} wf\text{-pat } (pat\text{-mset } p)$
shows $pat\text{-complete } C (pat\text{-mset } p) \longleftrightarrow (\forall q. (p,q) \in \Rightarrow_{nd} \longrightarrow pat\text{-complete } C$
 $(pat\text{-mset } q))$
 ⟨proof⟩

lemma *P-step-pp-trans*: **assumes** $(P,Q) \in \Rightarrow$
shows $pats\text{-mset } P \Rightarrow_s pats\text{-mset } Q$
 ⟨proof⟩

theorem *P-step-pcorrect*: **assumes** $wf: wf\text{-pats } (pats\text{-mset } P)$ **and** $step: (P,Q) \in \Rightarrow$
shows $wf\text{-pats } (pats\text{-mset } Q) \wedge (pats\text{-complete } C (pats\text{-mset } P) = pats\text{-complete } C$
 $(pats\text{-mset } Q))$
 ⟨proof⟩

corollary *P-steps-pcorrect*: **assumes** $wf: wf\text{-pats } (pats\text{-mset } P)$
and $step: (P,Q) \in \Rightarrow^*$
shows $wf\text{-pats } (pats\text{-mset } Q) \wedge (pats\text{-complete } C (pats\text{-mset } P) \longleftrightarrow pats\text{-complete } C$
 $(pats\text{-mset } Q))$
 ⟨proof⟩

lemma *nd-step-to-P-step*: **assumes** $(p,q) \in \Rightarrow_{nd}$
shows $\exists Q. add\text{-mset } p P \Rightarrow_m add\text{-mset } q Q$
 ⟨proof⟩

lemma *nd-steps-to-P-steps*: **assumes** $(p,q) \in \Rightarrow_{nd}^*$
shows $\exists Q. (\Rightarrow_m)^{**} (add\text{-}mset\ p\ P) (add\text{-}mset\ q\ Q)$
 $\langle proof \rangle$

lemma *P-step-to-nd-step*: **assumes** $P \Rightarrow_m Q$
and $q \in \# Q$ **shows** $\exists p \in \# P. (p,q) \in \Rightarrow_{nd}^=$
 $\langle proof \rangle$

lemma *P-steps-to-nd-steps*: **assumes** $(\Rightarrow_m)^{**} P Q$
and $q \in \# Q$ **shows** $\exists p \in \# P. (p,q) \in \Rightarrow_{nd}^*$
 $\langle proof \rangle$

lemma *nd-steps-fail-iff-Psteps-fail*: $(p, \{\#\}) \in \Rightarrow_{nd}^* \longleftrightarrow (\Rightarrow_m)^{**} \{\#p\}$ *bottom-mset*
 $\langle proof \rangle$

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 $Q = \{\#\} \wedge pats\text{-}complete\ C\ (pats\text{-}mset\ P)$ — either the result is and input P is complete
 $\vee Q = bottom\text{-}mset \wedge \neg pats\text{-}complete\ C\ (pats\text{-}mset\ P)$ — or the result = bot and P is not complete
 $\langle proof \rangle$

theorem *nd-pstep*:

assumes *non-improved*: $\neg improved$
and *wf*: *wf-pat* (*pat-mset* p)
shows $\neg pat\text{-}complete\ C\ (pat\text{-}mset\ p) \longleftrightarrow (p, \{\#\}) \in \Rightarrow_{nd}^*$
 $\langle proof \rangle$

theorem *P-step-improved*:

fixes $P :: ('f, 'v, 's)\ pats\text{-}problem\text{-}mset$
assumes *improved*
and *inf*: *infinite* (*UNIV* :: $'v\ set$)
and *wf*: *wf-pats* (*pats-mset* P) **and** *NF*: $(P,Q) \in \Rightarrow^!$
shows *pats-complete* $C\ (pats\text{-}mset\ P) \longleftrightarrow pats\text{-}complete\ C\ (pats\text{-}mset\ Q)$ — equivalence
 $p \in \# Q \implies finite\text{-}constr\text{-}form\text{-}pat\ C\ (pat\text{-}mset\ p)$ — all remaining problems are in finite-constr-form
 $\langle proof \rangle$

theorem *nd-step-improved*:

fixes $p :: ('f, 'v, 's)\ pat\text{-}problem\text{-}mset$
assumes *improved*
and *inf*: *infinite* (*UNIV* :: $'v\ set$)

```

and wf: wf-pat (pat-mset p)
shows (p,q) ∈ ⇒nd  $\widetilde{\sim}$  n ⇒ n ≤ num-syms-pat p * (num-syms-pat p * m + 2)
      (p,q) ∈ ⇒nd  $\widetilde{\sim}$  n ⇒ measure-pat-poly c q + n ≤ (c + num-syms-pat p) *
      (num-syms-pat p * m + 2)
      (p,q) ∈ ⇒nd! ⇒ finite-constr-form-pat C (pat-mset q)
      (p,q) ∈ ⇒nd* ⇒ pat-complete C (pat-mset p) ⇒ pat-complete C (pat-mset
q)
      ¬ pat-complete C (pat-mset p) ⇒ ∃ q. (p,q) ∈ ⇒nd! ∧ ¬ pat-complete C
      (pat-mset q)
      ¬ pat-complete C (pat-mset p) ⇔ (∃ q. (p,q) ∈ ⇒nd! ∧ ¬ pat-complete C
      (pat-mset q))
⟨proof⟩
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 $lvars\text{-}mp :: ('f, 'v, 's)\text{match-problem-mset} \Rightarrow 'v\ \text{set}$ **where**
 $lvars\text{-}mp\ mp = (\bigcup\ (\text{vars}\ 'snd\ 'mp\text{-}mset\ mp))$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where $\text{pats-mset-list} \equiv \text{mset} \circ \text{map} \text{ pat-mset-list}$

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

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

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**

$\text{empty-lr } mp = (\text{case } mp \text{ of } (lx, rx, -) \Rightarrow \text{lx} = [] \wedge \text{rx} = [])$

fun $\text{zipAll} :: 'a \text{ list} \Rightarrow 'b \text{ list list} \Rightarrow ('a \times 'b \text{ list}) \text{ list}$ **where**

$\text{zipAll } [] \text{ -} = []$
 $| \text{zipAll } (x \# xs) \text{ yss} = (x, \text{map } \text{hd } \text{yss}) \# \text{zipAll } xs (\text{map } \text{tl } \text{yss})$

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**

$\text{var-form-of-match-rx} = \text{map} (\text{map-prod } \text{id} (\text{map } \text{the-Var})) \circ \text{fst}$

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

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

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

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

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

$pat\text{-of}\text{-var}\text{-form}\text{-list} = map\ match\text{-of}\text{-var}\text{-form}\text{-list}$

lemma $size\text{-zip}[termination\text{-simp}]$: $length\ ts = length\ ls \implies size\text{-list}\ (\lambda p.\ size\ (snd\ p))\ (zip\ ts\ ls)$
 $< Suc\ (size\text{-list}\ size\ ls)$
 $\langle proof \rangle$

fun $match\text{-decomp}\text{-lin}\text{-impl} :: ('f,'v,'s)match\text{-problem}\text{-list} \Rightarrow ('f,'v,'s)match\text{-problem}\text{-lx}\ option$ **where**

$match\text{-decomp}\text{-lin}\text{-impl}\ [] = Some\ []$
 $| match\text{-decomp}\text{-lin}\text{-impl}\ ((Fun\ f\ ts,\ Fun\ g\ ls) \# mp) = (if\ (f,length\ ts) = (g,length\ ls)\ then$
 $match\text{-decomp}\text{-lin}\text{-impl}\ (zip\ ts\ ls\ @\ mp)\ else\ None)$
 $| match\text{-decomp}\text{-lin}\text{-impl}\ ((Var\ x,\ Fun\ g\ ls) \# mp) = (map\ option\ (Cons\ (x,\ Fun\ g\ ls))\ (match\text{-decomp}\text{-lin}\text{-impl}\ mp))$
 $| match\text{-decomp}\text{-lin}\text{-impl}\ ((t,\ Var\ y) \# mp) = match\text{-decomp}\text{-lin}\text{-impl}\ mp$

fun $pat\text{-inner}\text{-lin}\text{-impl} :: ('f,'v,'s)pat\text{-problem}\text{-list} \Rightarrow ('f,'v,'s)pat\text{-problem}\text{-lx} \Rightarrow ('f,'v,'s)pat\text{-problem}\text{-lx}\ option$ **where**

$pat\text{-inner}\text{-lin}\text{-impl}\ []\ pd = Some\ pd$
 $| pat\text{-inner}\text{-lin}\text{-impl}\ (mp \# p)\ pd = (case\ match\text{-decomp}\text{-lin}\text{-impl}\ mp\ of$
 $None \Rightarrow pat\text{-inner}\text{-lin}\text{-impl}\ p\ pd$
 $| Some\ mp' \Rightarrow if\ mp' = []\ then\ None$
 $else\ pat\text{-inner}\text{-lin}\text{-impl}\ p\ (mp' \# pd))$

fun $pairs\text{-of}\text{-list}$ **where**

$pairs\text{-of}\text{-list}\ (x \# y \# xs) = (x,y) \# pairs\text{-of}\text{-list}\ (y \# xs)$
 $| pairs\text{-of}\text{-list}\ - = []$

lemma $set\text{-pairs}\text{-of}\text{-list}$: $set\ (pairs\text{-of}\text{-list}\ xs) = \{ (xs\ !\ i,\ xs\ !\ (Suc\ i)) \mid i.\ Suc\ i < length\ xs \}$
 $\langle proof \rangle$

lemma $diff\text{-pairs}\text{-of}\text{-list}$: $(\exists\ x \in set\ xs.\ \exists\ y \in set\ xs.\ f\ x \neq f\ y) \longleftrightarrow$
 $(\exists\ (x,y) \in set\ (pairs\text{-of}\text{-list}\ xs).\ f\ x \neq f\ y)$ (**is** ?l = ?r)
 $\langle proof \rangle$

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

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

lemma $compute\text{-k}\text{-parameter}$: $\forall p \in set\ P.\ length\ p < compute\text{-k}\text{-parameter}\ P$
 $\langle proof \rangle$

lemma *compute-k-parameter-1: compute-k-parameter $P > 1$*
 ⟨proof⟩

context *pattern-completeness-context*
begin

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

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

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

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

definition *pat-lin-impl :: nat ⇒ ('f,'v,'s)pat-problem-list ⇒ ('f,'v,'s)pat-problem-list list option* **where**

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

partial-function (*tailrec*) *pats-lin-impl :: nat ⇒ ('f,'v,'s)pats-problem-list ⇒ bool* **where**

pats-lin-impl n ps = (case ps of [] ⇒ True
| p # ps1 ⇒ (case pat-lin-impl n p of

None \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' \neq []) (map (λ (y,ts'). (y, remdups ts'))
 new)
 in decomp'-main-loop (n + l) xs (cleaned @ rxs) out))

definition decomp'-impl **where**

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

definition apply-decompose' :: ('f,'v,'s)match-problem-lr \Rightarrow bool

where apply-decompose' mp = (improved \wedge (case mp of (xl,(rx,b)) \Rightarrow (\neg b \wedge xl = [])))

definition match-decomp'-impl :: nat \Rightarrow ('f,'v,'s)match-problem-list \Rightarrow (nat \times ('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 \Rightarrow ('f,'v,'s)pat-problem-list \Rightarrow ('f,'v,'s)pat-problem-lr \Rightarrow

($\text{nat} \times ('f, 'v, 's)\text{pat-problem-lr}$) option **where**
 $\text{pat-inner-impl } n \ [] \text{ pd} = \text{Some } (n, \text{pd})$
 $| \text{pat-inner-impl } n \ (\text{mp} \# \text{p}) \text{ pd} = (\text{case } \text{match-decomp}'\text{-impl } n \ \text{mp} \text{ of}$
 $\quad \text{None} \Rightarrow \text{pat-inner-impl } n \ \text{p} \ \text{pd}$
 $\quad | \text{Some } (n', \text{mp}') \Rightarrow \text{if empty-lr } \text{mp}' \text{ then None}$
 $\quad \quad \text{else } \text{pat-inner-impl } n' \ \text{p} \ (\text{mp}' \# \text{pd}))$

context

fixes $CC :: 'f \times 's \text{ list} \Rightarrow 's \text{ option}$

begin

definition $\text{pat-impl} :: \text{nat} \Rightarrow \text{nat} \Rightarrow ('f, 'v, 's)\text{pat-problem-list} \Rightarrow ('f, 'v, 's)\text{pat-impl-result}$
where

$\text{pat-impl } n \ \text{nl} \ \text{p} = (\text{case } \text{pat-inner-impl } \text{nl} \ \text{p} \ [] \text{ of } \text{None} \Rightarrow \text{New-Problems } (n, \text{nl}, [])$
 $\quad | \text{Some } (nl', \text{p}') \Rightarrow (\text{case } \text{partition } (\lambda \text{mp. } \text{snd } (\text{snd } \text{mp})) \ \text{p}' \text{ of}$
 $\quad \quad (\text{ivc}, \text{no-ivc}) \Rightarrow \text{if no-ivc} = [] \text{ then Incomplete} \text{ — detected inf-var-conflict (or}$
 empty mp)
 $\quad \quad \text{else (if improved} \wedge (\forall \text{mp} \in \text{set no-ivc. } \text{fst } \text{mp} = []) \text{ then}$
 $\quad \quad \quad \text{Fin-Constr-Form } (\text{map } (\text{map } \text{snd } \text{o } \text{fst } \text{o } \text{snd}) \ (\text{filter} \text{ — inf-var-conflict}' +$
 match-clash-sort
 $\quad \quad \quad (\lambda \text{mp. } \forall \text{xts} \in \text{set } (\text{fst } (\text{snd } \text{mp})). \text{is-singleton-list } (\text{map } (\mathcal{T}(CC, \mathcal{V})) \ (\text{snd}$
 $\text{xts}))) \text{ no-ivc}))$
 $\quad \quad \text{else (let } x = \text{find-var improved no-ivc; } \text{p}'l = \text{map mp-lr-list } \text{p}'$
 $\quad \quad \text{in New-Problems } (n + m, \text{nl}', \text{map } (\lambda \tau. \text{subst-pat-problem-list } \tau \ \text{p}'l) \ (\tau\text{s-list}$
 $\text{ } n \ x))))))$

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

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

definition $\text{pat-complete-impl} :: ('f, 'v, 's)\text{pats-problem-list} \Rightarrow \text{bool where}$

$\text{pat-complete-impl } \text{ps} = (\text{let}$
 $\quad n = \text{Suc } (\text{max-list } (\text{List.maps } (\text{map } \text{fst } \text{o } \text{vars-term-list } \text{o } \text{fst}) \ (\text{concat } (\text{concat}$
 $\text{ps})))));$
 $\quad \text{nl} = 0;$
 $\quad k = \text{compute-k-parameter } \text{ps};$
 $\quad \text{ps}' = \text{if improved then map } (\text{map } (\text{map } (\text{apsnd } (\text{map-vars } \text{renVar})))) \ \text{ps} \text{ else}$
 ps
 $\quad \text{in } \text{pats-impl } n \ \text{nl} \ \text{ps}'$

end

end

end

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

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*: $(\sum x \in \#A. image\ mset\ (f\ x)\ B) = (\sum i \in \#B. \{\#f\ x\ i.\ x \in \#A\})$
 <proof>

lemma *vars-mp-mset-add*: $vars\ mp\ mset\ (mp + mp') = vars\ mp\ mset\ mp + vars\ mp\ mset\ mp'$
 <proof>

zipAll

lemma *zipAll*: **assumes** $length\ as = n$
and $\bigwedge bs. bs \in set\ bss \implies length\ bs = n$
shows $zipAll\ as\ bss = map\ (\lambda i. (as\ !\ i, map\ (\lambda bs. bs\ !\ i)\ bss))\ [0..<n]$
 <proof>

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

lemma *tvars-pat-mono*: $P \subseteq P' \implies tvars\ pat\ P \subseteq tvars\ pat\ P'$
 <proof>

lemma *mset-concat-union*:
 $mset\ (concat\ xs) = \sum \# (mset\ (map\ mset\ xs))$
 <proof>

lemma *in-map-mset*[intro]:
 $a \in \#A \implies f\ a \in \# image\ mset\ f\ A$
 <proof>

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

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

lemma *mp-rx-append*: $\text{mp-rx } (xs \ @ \ ys, \ b) = \text{mp-rx } (xs,b) + \text{mp-rx } (ys,b)$
 $\langle \text{proof} \rangle$

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

lemma *set-tvars-match-list*: $\text{set } (\text{tvars-match-list } mp) = \text{tvars-match } (\text{set } mp)$
 $\langle \text{proof} \rangle$

lemma *set-tvars-pat-list*: $\text{set } (\text{tvars-pat-list } pp) = \text{tvars-pat } (\text{pat-list } pp)$
 $\langle \text{proof} \rangle$

lemma *non-uniq-image-diff*: $\neg \text{UNIQ } (\alpha \ ' \ \text{set } vs) \longleftrightarrow (\exists \ v \in \ \text{set } vs. \ \exists \ w \in \ \text{set}$
 $vs. \ \alpha \ v \neq \alpha \ w)$
 $\langle \text{proof} \rangle$

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**
 $wf\text{-ts } ts = (ts \neq [] \wedge \text{distinct } ts \wedge (\forall \ j < \ \text{length } ts. \ \forall \ i < \ j. \ \text{conflicts } (ts \ ! \ i) \ (ts$
 $\ ! \ j) \neq \text{None}))$

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

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

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

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

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

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**
 $wf-pat-lr\ mps = (Ball\ (set\ mps)\ (\lambda\ mp.\ wf-lr3\ mp \wedge \neg\ empty\text{-}lr\ mp))$

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

lemma $wf-rx\text{-}mset$: **assumes** $mset\ rx = mset\ rx'$
shows $wf-rx\ (rx, b) = wf-rx\ (rx', b)$
 $\langle proof \rangle$

lemma $wf-rx2\text{-}mset$: **assumes** $mset\ rx = mset\ rx'$
shows $wf-rx2\ (rx, b) = wf-rx2\ (rx', b)$
 $\langle proof \rangle$

lemma $wf-lr2\text{-}mset$: **assumes** $mset\ rx = mset\ rx'$
shows $wf-lr2\ (lx, (rx, b)) = wf-lr2\ (lx, (rx', b))$
 $\langle proof \rangle$

lemma $mp\text{-}lr\text{-}mset$: **assumes** $mset\ rx = mset\ rx'$
shows $mp\text{-}lr\ (lx, (rx, b)) = mp\text{-}lr\ (lx, (rx', b))$
 $\langle proof \rangle$

lemma $mp\text{-}list\text{-}lr$: $mp\text{-}list\ (mp\text{-}lr\text{-}list\ mp) = mp\text{-}lr\ mp$
 $\langle proof \rangle$

lemma $pat\text{-}mset\text{-}list\text{-}lr$: $pat\text{-}mset\text{-}list\ (map\ mp\text{-}lr\text{-}list\ pp) = pat\text{-}lr\ pp$

$\langle \text{proof} \rangle$

lemma *size-term-0[simp]*: $\text{size } (t :: (f, 'v)\text{term}) > 0$
 $\langle \text{proof} \rangle$

lemma *wf-ts-no-conflict-alt-def*: $(\forall j < \text{length } ts. \forall i < j. \text{conflicts } (ts ! i) (ts ! j) \neq \text{None})$
 $\longleftrightarrow (\forall s t. s \in \text{set } ts \longrightarrow t \in \text{set } ts \longrightarrow \text{conflicts } s t \neq \text{None})$ (**is** ?l = ?r)
 $\langle \text{proof} \rangle$

Continue with properties of the sub-algorithms

lemma *insert-rx: assumes* $\text{res}: \text{insert-rx } t \ x \ \text{rx}b = \text{res}$
and $\text{wf}: \text{wf-rx } \text{rx}b$
and $\text{mp}: \text{mp} = (ls, \text{rx}b)$
shows $\text{res} = \text{Some } \text{rx}' \implies (\rightarrow_m)^{**} (\text{add-mset } (t, \text{Var } x) (\text{mp-lr } \text{mp} + M)) (\text{mp-lr } (ls, \text{rx}') + M) \wedge \text{wf-rx } \text{rx}'$
 $\wedge \text{lvars-mp } (\text{add-mset } (t, \text{Var } x) (\text{mp-lr } \text{mp} + M)) \supseteq \text{lvars-mp } (\text{mp-lr } (ls, \text{rx}') + M)$
 $\text{res} = \text{None} \implies \text{match-fail } (\text{add-mset } (t, \text{Var } x) (\text{mp-lr } \text{mp} + M))$
 $\langle \text{proof} \rangle$

lemma *decomp-impl*: $\text{decomp-impl } \text{mp} = \text{res} \implies$
 $(\text{res} = \text{Some } \text{mp}' \longrightarrow (\rightarrow_m)^{**} (\text{mp-list } \text{mp} + M) (\text{mp-lr } \text{mp}' + M) \wedge \text{wf-lr } \text{mp}'$
 $\wedge \text{lvars-mp } (\text{mp-list } \text{mp} + M) \supseteq \text{lvars-mp } (\text{mp-lr } \text{mp}' + M))$
 $\wedge (\text{res} = \text{None} \longrightarrow (\exists \text{mp}'. (\rightarrow_m)^{**} (\text{mp-list } \text{mp} + M) \text{mp}' \wedge \text{match-fail } \text{mp}'))$
 $\langle \text{proof} \rangle$

lemma *match-decomp-lin-impl*: $\text{match-decomp-lin-impl } \text{mp} = \text{res} \implies \text{ll-mp } (\text{mp-list } \text{mp} + M) \implies$
 $(\text{res} = \text{Some } \text{mp}' \longrightarrow (\rightarrow_m)^{**} (\text{mp-list } \text{mp} + M) (\text{mp-list } (\text{mp-lx } \text{mp}') + M) \wedge$
 $\text{wf-lx } \text{mp}' \wedge \text{ll-mp } (\text{mp-list } (\text{mp-lx } \text{mp}') + M))$
 $\wedge (\text{res} = \text{None} \longrightarrow (\exists \text{mp}'. (\rightarrow_m)^{**} (\text{mp-list } \text{mp} + M) \text{mp}' \wedge \text{match-fail } \text{mp}'))$
 $\langle \text{proof} \rangle$

lemma *pat-inner-lin-impl*: **assumes** $\text{pat-inner-lin-impl } p \ \text{pd} = \text{res}$
and $\text{wf-pat-lx } \text{pd} \ \forall \text{mp} \in \text{set } p. \text{ll-mp } (\text{mp-list } \text{mp})$
and $\text{tvars-pat } (\text{pat-mset } (\text{pat-mset-list } p + \text{pat-lx } \text{pd})) \subseteq V$
shows $\text{res} = \text{None} \implies (\text{add-mset } (\text{pat-mset-list } p + \text{pat-lx } \text{pd}) P, P) \in \Rightarrow^+$
and $\text{res} = \text{Some } p' \implies (\text{add-mset } (\text{pat-mset-list } p + \text{pat-lx } \text{pd}) P, \text{add-mset } (\text{pat-lx } p') P) \in \Rightarrow^*$
 $\wedge \text{wf-pat-lx } p' \wedge \text{tvars-pat } (\text{pat-mset } (\text{pat-lx } p')) \subseteq V$
 $\langle \text{proof} \rangle$

lemma *pat-mset-list*: $\text{pat-mset } (\text{pat-mset-list } p) = \text{pat-list } p$
 $\langle \text{proof} \rangle$

lemma *vars-mp-mset-subst*: $\text{vars-mp-mset} (\text{mp-list} (\text{subst-match-problem-list } \tau \text{ mp}))$
 $= \text{vars-mp-mset} (\text{mp-list } \text{mp})$
 $\langle \text{proof} \rangle$

lemma *subst-conversion*: $\text{map} (\lambda \tau. \text{subst-pat-problem-mset } \tau (\text{pat-mset-list } p)) \text{ xs}$
 $=$
 $\text{map pat-mset-list} (\text{map} (\lambda \tau. \text{subst-pat-problem-list } \tau p) \text{ xs})$
 $\langle \text{proof} \rangle$

lemma *ll-mp-subst*: $\text{ll-mp} (\text{mp-list} (\text{subst-match-problem-list } \tau \text{ mp})) = \text{ll-mp} (\text{mp-list } \text{mp})$
 $\langle \text{proof} \rangle$

lemma *ll-pp-subst*: $\text{ll-pp} (\text{subst-pat-problem-list } \tau p) = \text{ll-pp } p$
 $\langle \text{proof} \rangle$

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

lemma *pat-lin-impl*:
assumes $\text{pat-lin-impl } n p = \text{res}$
and $\text{vars: tvars-pat} (\text{pat-list } p) \subseteq \{..<n\} \times S$
and $\text{linear: ll-pp } p$
shows $\text{res} = \text{None} \implies (\text{add-mset} (\text{pat-mset-list } p) P, \text{add-mset} \{\#\} P) \in \Rightarrow^*$
and $\text{res} = \text{Some } ps \implies (\text{add-mset} (\text{pat-mset-list } p) P, \text{mset} (\text{map pat-mset-list } ps) + P) \in \Rightarrow^+$
 $\wedge \text{tvars-pat} (\bigcup (\text{pat-list } ' \text{ set } ps)) \subseteq \{..<n+m\} \times S$
 $\wedge \text{Ball} (\text{set } ps) \text{ ll-pp}$
 $\langle \text{proof} \rangle$

lemma *pats-mset-list*: $\text{pats-mset} (\text{pats-mset-list } ps) = \text{pat-list } ' \text{ set } ps$
 $\langle \text{proof} \rangle$

lemma *pats-lin-impl*: **assumes** $\forall p \in \text{set } ps. \text{tvars-pat} (\text{pat-list } p) \subseteq \{..<n\} \times S$
and $\text{Ball} (\text{set } ps) \text{ ll-pp}$
and $\forall pp \in \text{pat-list } ' \text{ set } ps. \text{wf-pat } pp$
shows $\text{pats-lin-impl } n ps = \text{pats-complete } C (\text{pat-list } ' \text{ set } ps)$
 $\langle \text{proof} \rangle$

corollary *pat-complete-lin-impl*:
assumes $\text{wf: snd } ' \bigcup (\text{vars } ' \text{ fst } ' \text{ set } (\text{concat} (\text{concat } P))) \subseteq S$
and $\text{left-linear: Ball} (\text{set } P) \text{ ll-pp}$
shows $\text{pat-complete-lin-impl} (P :: ('f, 'v, 's)\text{pats-problem-list}) \longleftrightarrow \text{pats-complete } C (\text{pat-list } ' \text{ set } P)$
 $\langle \text{proof} \rangle$

lemma *match-var-impl*: **assumes** *wf*: *wf-lr mp*
and *match-var-impl mp = (xs, mpFin)*
shows $(\rightarrow_m)^{**} (mp\text{-lr } mp) (mp\text{-lr } mpFin)$
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>

lemma *match-steps-impl*: **assumes** *match-steps-impl mp = res*
shows *res = Some (xs, mp \wedge) \implies $(\rightarrow_m)^{**} (mp\text{-list } mp) (mp\text{-lr } mp') \wedge wf\text{-lr2 } mp'$*
 \wedge *lvars-mp (mp-list mp) \supseteq lvars-mp (mp-lr mp')*
 \wedge *set xs = lvars-mp (mp-list (mp-lx (fst mp')))*
and *res = None \implies \exists mp'. $(\rightarrow_m)^{**} (mp\text{-list } mp) mp' \wedge match\text{-fail } mp'$*
<proof>

lemma *finite-sort-imp-finite-sort-vars*:
assumes *t : σ in $\mathcal{T}(C, \mathcal{V})$*
and *x \in vars t*
and \neg *inf-sort σ*
shows \neg *inf-sort (snd x)*
<proof>

context
fixes *renVar :: 'v \Rightarrow 'v*
and *renNat :: nat \Rightarrow 'v*
and *fcf-solve :: nat \Rightarrow ('f, 's)spp-list \Rightarrow bool*
and *CC :: 'f \times 's list \Rightarrow 's option*
assumes *renaming-ass: improved \implies renaming-funs renNat renVar*
and *fcf-solve: improved \implies fcf-solver k fcf-solve*
and *CC: improved \implies CC = C*
begin

abbreviation *Match-decomp'-impl where Match-decomp'-impl \equiv match-decomp'-impl renNat*

abbreviation *Decomp'-main-loop where Decomp'-main-loop \equiv decomp'-main-loop renNat*

abbreviation *Decomp'-impl where Decomp'-impl \equiv decomp'-impl renNat*

abbreviation *Pat-inner-impl where Pat-inner-impl \equiv pat-inner-impl renNat*

abbreviation *Pat-impl where Pat-impl \equiv pat-impl renNat CC*

abbreviation *Pats-impl where Pats-impl \equiv pats-impl renNat fcf-solve CC*

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

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

definition *lvar-cond where lvar-cond n V = (V \subseteq allowed-vars n)*

definition *lvar-cond-mp where lvar-cond-mp n mp = lvar-cond n (lvars-mp mp)*

definition *lvar-cond-pp where lvar-cond-pp n pp = lvar-cond n (lvars-pp pp)*

definition *size-cond-pp* **where** *size-cond-pp* $pp = (\text{size } pp < k)$

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

$\text{lvar-cond } n (\text{insert } x A) = (x \in \text{allowed-vars } n \wedge \text{lvar-cond } n A)$
 $\text{lvar-cond } n \{\}$
 $\text{lvar-cond } n (A \cup B) = (\text{lvar-cond } n A \wedge \text{lvar-cond } n B)$
 $\text{lvar-cond } n (\bigcup As) = (\forall A \in As. \text{lvar-cond } n A)$
 $\langle \text{proof} \rangle$

lemma *lvar-cond-mono*: $n \leq n' \implies \text{lvar-cond } n V \implies \text{lvar-cond } n' V$
 $\langle \text{proof} \rangle$

lemma *pair-fst-imageI*: $(a,b) \in c \implies a \in \text{fst } ' c \langle \text{proof} \rangle$

lemma *not-in-fstD*: $x \notin \text{fst } ' a \implies \forall z. (x,z) \notin a \langle \text{proof} \rangle$

lemma *many-remdups-steps*: **assumes** $\text{mp-mset } mp2 = \text{mp-mset } mp1 \text{ } mp2 \subseteq \#$
 $mp1$
shows $(\rightarrow_m)^{**} mp1 \text{ } mp2$
 $\langle \text{proof} \rangle$

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 $(\rightarrow_m)^{**} (mp1 + mp2) \text{ } mp2$
 $\langle \text{proof} \rangle$

lemma *decomp'-impl*: **assumes**

$\text{wf-lr2 } mp$
 $\text{set } xs = \text{lvars-mp } (mp\text{-list } (mp\text{-lx } (\text{fst } mp)))$
 $\text{lvar-cond-mp } n (mp\text{-lr } mp)$
 $\text{Decomp}'\text{-impl } n \text{ } xs \text{ } mp = (n', mp')$
 improved
shows $\text{wf-lr3 } mp'$
 $\text{lvar-cond-mp } n' (mp\text{-lr } mp')$
 $(\rightarrow_m)^{**} (mp\text{-lr } mp) (mp\text{-lr } mp')$
 $n \leq n'$
 $\langle \text{proof} \rangle$

lemma *match-decomp'-impl*: **assumes** $\text{Match-decomp}'\text{-impl } n \text{ } mp = \text{res}$

and $\text{lvc: lvar-cond-mp } n (mp\text{-list } mp)$
shows $\text{res} = \text{Some } (n', mp') \implies (\rightarrow_m)^{**} (mp\text{-list } mp) (mp\text{-lr } mp') \wedge \text{wf-lr3 } mp'$
 $\wedge \text{lvar-cond-mp } n' (mp\text{-lr } mp') \wedge n \leq n'$
and $\text{res} = \text{None} \implies \exists mp'. (\rightarrow_m)^{**} (mp\text{-list } mp) \text{ } mp' \wedge \text{match-fail } mp'$
 $\langle \text{proof} \rangle$

lemma *pat-inner-impl*: **assumes** *Pat-inner-impl* n p $pd = res$
and *wf-pat-lr* pd
and *tvars-pat* ($pat\text{-}mset$ ($pat\text{-}mset\text{-}list$ $p + pat\text{-}lr$ pd)) $\subseteq V$
and *lvar-cond-pp* n ($pat\text{-}mset\text{-}list$ $p + pat\text{-}lr$ pd) \wedge *size-cond-pp* ($pat\text{-}mset\text{-}list$ $p + pat\text{-}lr$ pd)
shows $res = None \implies (add\text{-}mset$ ($pat\text{-}mset\text{-}list$ $p + pat\text{-}lr$ pd) $P, P) \in \Rightarrow^+$
and $res = Some$ (n', p') $\implies (add\text{-}mset$ ($pat\text{-}mset\text{-}list$ $p + pat\text{-}lr$ pd) $P, add\text{-}mset$ ($pat\text{-}lr$ p') $P) \in \Rightarrow^*$
 \wedge *wf-pat-lr* $p' \wedge$ *tvars-pat* ($pat\text{-}mset$ ($pat\text{-}lr$ p')) $\subseteq V$
 \wedge *lvar-cond-pp* n' ($pat\text{-}lr$ p') \wedge *size-cond-pp* ($pat\text{-}lr$ p') $\wedge n \leq n'$
 $\langle proof \rangle$

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

lemma *pat-impl*:
assumes *Pat-impl* n nl $p = res$
and *vars*: *tvars-pat* ($pat\text{-}list$ p) $\subseteq \{..<n\} \times S$
and *lvarsAll*: $\forall pp \in \#$ $add\text{-}mset$ ($pat\text{-}mset\text{-}list$ p) P . *lvar-cond-pp* nl $pp \wedge$ *size-cond-pp* pp
shows $res = Incomplete \implies (add\text{-}mset$ ($pat\text{-}mset\text{-}list$ p) $P, add\text{-}mset$ $\{\#\}$ $P) \in \Rightarrow^*$
and $res = New\text{-}Problems$ (n', nl', ps) $\implies (add\text{-}mset$ ($pat\text{-}mset\text{-}list$ p) $P, mset$ (map $pat\text{-}mset\text{-}list$ ps) $+ P) \in \Rightarrow^+$
 \wedge *tvars-pat* (\bigcup ($pat\text{-}list$ ' set ps)) $\subseteq \{..<n'\} \times S$
 \wedge ($\forall pp \in \#$ $mset$ (map $pat\text{-}mset\text{-}list$ ps) $+ P$. *lvar-cond-pp* nl' $pp \wedge$ *size-cond-pp* pp) $\wedge n \leq n'$
and $res = Fin\text{-}Constr\text{-}Form$ $fcf \implies improved \wedge (\exists P'. (add\text{-}mset$ ($pat\text{-}mset\text{-}list$ p) $P, add\text{-}mset$ $P' P) \in \Rightarrow^*$
 \wedge *finite-constructor-form-pat* ($set3$ fcf)
 \wedge *tvars-spat* ($set3$ fcf) $\subseteq \{..<n\} \times S$
 \wedge *length* $fcf < k$
 \wedge *pat-complete* C ($pat\text{-}mset$ P') = *simple-pat-complete* C SS ($set3$ fcf)
 $\langle proof \rangle$

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* ($pat\text{-}list$ p) $\subseteq \{..<n\} \times S$
and $\forall pp \in set$ ps . *lvar-cond-pp* nl ($pat\text{-}mset\text{-}list$ pp) \wedge *size-cond-pp* ($pat\text{-}mset\text{-}list$ pp)
and $\forall pp \in pat\text{-}list$ ' set ps . *wf-pat* pp
shows *Pats-impl* n nl $ps = pats\text{-}complete$ C ($pat\text{-}list$ ' set ps)
 $\langle proof \rangle$

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

corollary *pat-complete-impl*:
assumes *wf*: snd ' \bigcup ($vars$ ' fst ' set ($concat$ ($concat$ P))) $\subseteq S$

and k : $k = \text{compute-}k\text{-parameter } P$
shows $\text{Pat-complete-impl } (P :: ('f, 'v, 's)\text{pats-problem-list}) \longleftrightarrow \text{pats-complete } C$
 $(\text{pat-list } \text{' set } P)$
 $\langle \text{proof} \rangle$
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*: $\text{distinct } (\text{map fst } l) \implies x : \sigma \text{ in map-of } l \longleftrightarrow (x, \sigma) \in \text{set } l$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *sorts-of-ssig-list*:
assumes $((f, \sigma s), \tau) \in \text{set } Cs$
shows $\text{set } \sigma s \subseteq \text{set } (\text{sorts-of-ssig-list } Cs) \ \tau \in \text{set } (\text{sorts-of-ssig-list } Cs)$
 $\langle \text{proof} \rangle$

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

lemma *max-arity-list*:
 $((f, \sigma s), \tau) \in \text{set } Cs \implies \text{length } \sigma s \leq \text{max-arity-list } Cs$
 $\langle \text{proof} \rangle$

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

lemma *nonempty-sort*: $\bigwedge \sigma. \sigma \in \text{set } (\text{sorts-of-ssig-list } Cs) \implies \neg \text{empty-sort } (\text{map-of } Cs) \ \sigma$
 $\langle \text{proof} \rangle$

lemma *compute-inf-sorts*: $\sigma \in \text{compute-inf-sorts } Cs \iff \neg \text{finite-sort } (\text{map-of } Cs)$

σ
 <proof>

lemma *compute-card-sorts*: $\text{snd } (\text{compute-inf-card-sorts-bnd } k \text{ } Cs) = \text{min } k \text{ o } \text{card-of-sort } (\text{map-of } Cs)$

<proof>

sublocale *pattern-completeness-context-with-assms*

improved set (sorts-of-ssig-list Cs) map-of Cs max-arity-list Cs constr-list Cs

$\lambda s. s \in \text{compute-inf-sorts } Cs$

$\text{snd } (\text{compute-inf-card-sorts-bnd } k \text{ } Cs)$

for *improved*

<proof>

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 $m :: \text{nat}$ — upper bound on arities of constructors

and $Cl :: 's \Rightarrow ('f \times 's \text{ list}) \text{ list}$ — a function to compute all constructors of given sort as list

and $Is :: 's \Rightarrow \text{bool}$ — a function to indicate whether a sort is infinite

and $Cd :: 's \Rightarrow \text{nat}$ — a function to compute finite cardinality of sort

begin

definition *pat-complete-impl-fscd* = *pattern-completeness-context.pat-complete-impl*
 $m \ Cl \ Is \ \text{False} \ \text{undefined} \ \text{undefined} \ \text{undefined} \ \text{undefined}$

definition *pats-impl-fscd* = *pattern-completeness-context.pats-impl* $m \ Cl \ Is \ \text{False}$
 $\text{undefined} \ \text{undefined} \ \text{undefined}$

definition *pat-impl-fscd* = *pattern-completeness-context.pat-impl* $m \ Cl \ Is \ \text{False}$
 $\text{undefined} \ \text{undefined}$

definition *pat-inner-impl-fscd* = *pattern-completeness-context.pat-inner-impl* $Is \ \text{False}$
 undefined

definition *match-decomp'-impl-fscd* = *pattern-completeness-context.match-decomp'-impl*
 $Is \ \text{False} \ \text{undefined}$

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

find-var-fscd $p = (\text{case } \text{List.maps } (\lambda (lx, -). \text{lx}) \text{ } p \text{ of}$

$(x, t) \# - \Rightarrow x$

$| [] \Rightarrow (\text{let } (-, rx, b) = \text{hd } p$

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*
 ⟨*proof*⟩

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* ⟨*proof*⟩

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)*))
 ⟨*proof*⟩

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* × *'s list*) ⇒ *'s option*
and *rn* :: *nat* ⇒ *'v*
and *rv* :: *'v* ⇒ *'v*
and *fcf-solve* :: *nat* ⇒ (*f, 's*)*spp-list* ⇒ *bool*

begin

definition *pat-complete-impl-fcf* = *pattern-completeness-context.pat-complete-impl*
m Cl Is True rn rv fcf-solve C

definition *pats-impl-new* = *pattern-completeness-context.pats-impl m Cl Is True*
rn fcf-solve C

definition *pat-impl-new* = *pattern-completeness-context.pat-impl m Cl Is True rn*
C

definition *pat-inner-impl-new* = *pattern-completeness-context.pat-inner-impl Is True*
rn

definition *match-decomp'-impl-new* = *pattern-completeness-context.match-decomp'-impl*
Is True rn

definition *find-var-new* = *find-var True*

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

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

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

lemma *match-decomp'-impl-new-code* [code]:
 ⟨*match-decomp'-impl-new n mp* =
map-option ($\lambda(xs, mp). \text{if case } mp \text{ of } (xl, rx, b) \Rightarrow \neg b \wedge xl = [] \text{ then } pat-$
*tern-completeness-context.decomp'-impl rn n xs mp else } (n, mp))
 (*pattern-completeness-context.match-steps-impl Is mp*)
 ⟨*proof*⟩*

lemma *find-var-new-code*[code]:
 ⟨*find-var-new p* = *fst* (*hd* (*List.maps* ($\lambda(lx, rx). lx$) *p*))
 ⟨*proof*⟩

lemma *pat-inner-impl-new-code* [code]:
 ⟨*pat-inner-impl-new n [] pd* = *Some* (*n*, *pd*)
 ⟨*pat-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*))
 ⟨*proof*⟩

end

end

definition *decide-pat-complete-fscd* :: ((*f* × *'s list*) × *'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 ($\lambda s. s \in IS$)) *P*

definition *decide-pat-complete-lin* :: ((*f* × *s* list) × *s*)list ⇒ (*f*,*v*,*s*)pats-problem-list ⇒ bool **where**

decide-pat-complete-lin *Cs* *P* = (let
m = *max-arity-list* *Cs*;
Cl = *constr-list* *Cs*
in *pattern-completeness-context.pat-complete-lin-impl* *m* *Cl* *P*)

theorem *decide-pat-complete-lin*:

assumes *dist*: *distinct* (*map fst Cs*)
and *non-empty-sorts*: *decide-nonempty-sorts* (*sorts-of-ssig-list Cs*) *Cs* = *None*
and *P*: *snd* ' ∪ (*vars* ' *fst* ' *set* (*concat* (*concat P*))) ⊆ *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*⟩

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* ' ∪ (*vars* ' *fst* ' *set* (*concat* (*concat P*))) ⊆ *set* (*sorts-of-ssig-list Cs*)
shows *decide-pat-complete-fscd Cs P* = *pats-complete* (*map-of Cs*) (*pat-list* ' *set P*)
⟨*proof*⟩

definition *decide-pat-complete-fcf* :: - ⇒ - ⇒ - ⇒ ((*f* × *s* list) × *s*)list ⇒ (*f*,*v*,*s*)pats-problem-list ⇒ 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* (λ *s*. *s* ∈ *IS*) (*Mapping.lookup Cm*) *rn* *rv*
fcf-solve P

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

[[*y*. (*t*' , *Var y*) ← *pp*, *t*' = *t*]. *t* ← *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 '  $\bigcup$  (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>

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 =  $\iota$ )
<proof>

```

```

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 = Var (x,  $\iota$ )  $\wedge$  l = Var y  $\wedge$  finite-sort C  $\iota$   $\wedge$ 
  ( $\forall$  z. (Var z, Var y)  $\in$  mp  $\longrightarrow$  snd z =  $\iota$ )
<proof>

```

lemma *finite-var-form-imp-of-var-form-pat*:
finite-var-form-pat C pp \implies *var-form-pat pp*
 ⟨proof⟩

lemma *finite-var-form-pat-UNIQ-sort*:
assumes *fuf*: *finite-var-form-pat C pp*
and *f*: *f* \in *var-form-of-pat pp*
shows *UNIQ* (*snd* ‘ *f x*)
 ⟨proof⟩

lemma *finite-var-form-pat-pat-complete*:
assumes *fuf*: *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⟩

context *pattern-completeness-context*
begin

fun *flatten-triv-sort-main* :: (*f*,*nat* \times '*s*)*term* \Rightarrow (*f*,*nat* \times '*s*)*term* \times '*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* \Rightarrow if *cd-sort s* = 1 then (*Var* (0,*s*), *s*) else
 (*Fun f* (*map fst tss*), *s*))

definition *flatten-triv-sort* :: (*f*,*nat* \times '*s*)*term* \Rightarrow (*f*,*nat* \times '*s*)*term* **where**
flatten-triv-sort = *fst o flatten-triv-sort-main*

definition *flatten-triv-sort-pat* :: (*f*, '*s*)*simple-pat-problem* \Rightarrow (*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*) \longleftrightarrow *simple-pat-complete C*
SS p
 ⟨proof⟩

lemma *eliminate-uniq-spp*: **assumes** *finite-constructor-form-pat p*

and $p = \text{insert } mp \ p'$
and $mp = \text{insert } eqc \ mp'$
and $pn = \text{insert } mp' \ p'$
and $\text{UNIQ } eqc$
shows $\text{simple-pat-complete } C \ SS \ p \longleftrightarrow \text{simple-pat-complete } C \ SS \ pn$
 $\text{finite-constructor-form-pat } pn$
 $\langle \text{proof} \rangle$

lemma *decompose-spp*: **assumes** $\text{finite-constructor-form-pat } p$
and $p: p = \text{insert } mp \ p'$
and $mp: mp = \text{insert } eqc \ mp'$
and $\text{root}: \bigwedge t. t \in eqc \implies \text{root } t = \text{Some } (f, n)$
and $\text{eqcn}: eqcn = (\lambda i. (\lambda t. \text{args } t \ ! \ i) \ 'eqc) \ '\{0..<n\}$
and $pn: pn = (\text{if } \text{Ball } eqcn \ (\lambda eq. \text{UNIQ } (\mathcal{T}(C, \mathcal{V}) \ 'eq)) \ \text{then } \text{insert } (eqcn \cup \ mp') \ p' \ \text{else } p')$
shows $\text{simple-pat-complete } C \ SS \ p \longleftrightarrow \text{simple-pat-complete } C \ SS \ pn$
 $\text{finite-constructor-form-pat } pn$
 $\langle \text{proof} \rangle$

lemma *eliminate-clash-spp*: **assumes** $\text{finite-constructor-form-pat } p$
and $p = \text{insert } mp \ pn$
and $mp = \text{insert } eqc \ mp'$
and $eqc = \{s, t\} \cup eqc'$
and $\text{Conflict-Clash } s \ t$
shows $\text{simple-pat-complete } C \ SS \ p \longleftrightarrow \text{simple-pat-complete } C \ SS \ pn$
 $\text{finite-constructor-form-pat } pn$
 $\langle \text{proof} \rangle$

lemma *detect-sat-spp*:
assumes $p = \text{insert } \{\} \ p'$
shows $\text{simple-pat-complete } C \ SS \ p$
 $\langle \text{proof} \rangle$

lemma *detect-unsat-spp*:
shows $\neg \text{simple-pat-complete } C \ SS \ \{\}$
 $\langle \text{proof} \rangle$

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

$\text{Ball } Pn \ \text{finite-constructor-form-pat}$
 $\langle \text{proof} \rangle$

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

Ball Pn *finite-constructor-form-pat*
 $\langle \text{proof} \rangle$

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

Ball Pn *finite-constructor-form-pat*
 $\langle \text{proof} \rangle$

lemma *typed-restrict-imp-typed*: $t : s \ \text{in } \mathcal{T}(D, W \mid ' \ F) \implies t : s \ \text{in } \mathcal{T}(D, W)$
 $\langle \text{proof} \rangle$

lemma *eliminate-large-sort*: **assumes**
 $cond: \bigwedge i. \ i \leq (n :: nat) \implies snd \ (x \ i) = \iota \wedge eq \ i \in mp \ i \wedge Var \ (x \ i) \neq t \ i \wedge$
 $\{Var \ (x \ i), t \ i\} \subseteq eq \ i$ **and**
 $vars: x \ ' \ \{0..n\} \cap tvars\ spat \ p = \{\}$ **and**
 $large: card \ (t \ ' \ \{0..n\}) < card\ of\ sort \ C \ \iota$ **and**
 $fin: \text{finite-constructor-form-pat} \ (p \cup mp \ ' \ \{0..n\})$
shows *simple-pat-complete* $C \ SS \ (p \cup mp \ ' \ \{0..n\}) = \text{simple-pat-complete } C \ SS \ p$
 $\langle \text{proof} \rangle$

end

end

theory *FCF-Multiset*

imports

FCF-Set

Pattern-Completeness-Multiset

begin

fun *depth-gterm* :: $('f, 'v)\text{term} \Rightarrow nat$ **where**
 $\text{depth-gterm} \ (Fun \ f \ ts) = Suc \ (\text{max-list} \ (\text{map} \ \text{depth-gterm} \ ts))$
 $|\ \text{depth-gterm} \ - = Suc \ 0$

lemma *depth-gterm-arg*: $t \in set \ ts \implies \text{depth-gterm} \ t < \text{depth-gterm} \ (Fun \ f \ ts)$
 $\langle \text{proof} \rangle$

type-synonym $(f, 's)$ simple-match-problem-ms = $(f, \text{nat} \times 's)$ 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

$mset2 \equiv \text{image set-mset o set-mset}$

abbreviation $mset3 :: (f, 's)$ simple-pat-problem-ms \Rightarrow $(f, 's)$ simple-pat-problem where

$mset3 \equiv \text{image mset2 o set-mset}$

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)

<proof>

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: add-mset (add-mset t (add-mset t eqc)) mp \rightarrow_{ss} add-mset (add-mset t eqc) mp

| smp -singleton: add-mset {# t #} mp \rightarrow_{ss} mp

| smp -triv-sort: $t : \iota$ in $\mathcal{T}(C, \mathcal{V}) \Rightarrow \text{cd-sort } \iota = 1 \Rightarrow \text{add-mset (add-mset t eq) mp} \rightarrow_{ss} \text{mp}$

| smp -decomp: $(\bigwedge t. t \in \text{set-mset eqc} \Rightarrow \text{root } t = \text{Some } (f, n))$

$\Rightarrow \text{eqcn} = \{\#\{\#\text{args } t ! i. t \in \#\ \text{eqc}\#\}. i \in \#\ \text{mset } [0..<n]\#\}$

$\Rightarrow (\bigwedge \text{eq}. \text{eq} \in \text{set-mset eqcn} \Rightarrow \text{UNIQ } (\mathcal{T}(C, \mathcal{V}) \text{ ' set-mset eq}))$

$\Rightarrow \text{add-mset eqc mp} \rightarrow_{ss} \text{eqcn} + \text{mp}$

inductive smp -fail-ms :: $(f, 's)$ simple-match-problem-ms \Rightarrow bool **where**

smp -clash: Conflict-Clash s t \Rightarrow

$s \in \text{eqc} \Rightarrow t \in \text{eqc} \Rightarrow \text{eqc} \in \text{mset2 mp} \Rightarrow \text{smp-fail-ms mp}$

| smp -decomp-fail: $(\bigwedge t. t \in \text{set-mset eqc} \Rightarrow \text{root } t = \text{Some } (f, n))$

$\Rightarrow i < n$

$\Rightarrow \neg \text{UNIQ } (\mathcal{T}(C, \mathcal{V}) \text{ ' } (\lambda t. \text{args } t ! i) \text{ ' set-mset eqc})$

$\Rightarrow \text{smp-fail-ms (add-mset eqc mp)}$

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: *add-mset* {#} *p* \Rightarrow_{ss} {#}
| *spp-simp*: *mp* \rightarrow_{ss} *mp'* \Longrightarrow *add-mset mp p* \Rightarrow_{ss} {#*add-mset mp' p*#}
| *spp-delete*: *smp-fail-ms mp* \Longrightarrow *add-mset mp p* \Rightarrow_{ss} {#*p*#}
| *spp-delete-large-sort*:
($\bigwedge i. i \leq (n :: \text{nat}) \Longrightarrow \text{snd } (x \ i) = \iota \wedge \text{eq } i \in \# \text{ mp } i \wedge \text{Var } (x \ i) \neq t \ i \wedge \{ \text{Var } (x \ i), t \ i \} \subseteq \text{set-mset } (\text{eq } i) \Longrightarrow$
 $x \ ' \{0..n\} \cap \text{tvars-spat } (\text{mset3 } p) = \{ \} \Longrightarrow$
 $(\text{card } (t \ ' \{0..n\}) < \text{card-of-sort } C \ \iota) \Longrightarrow$
 $p + \text{mset } (\text{map } \text{mp } [0..< \text{Suc } n]) \Rightarrow_{ss} \{ \# \ p \ \# \}$
| *spp-inst*: {# {# *Var x*, *t* #} #} $\in \# \ p$
 \Longrightarrow *is-Fun t*
 \Longrightarrow *fst* ' *tvars-spat* (*mset3 p*) $\cap \{n..<n + m\} = \{ \}$
 \Longrightarrow *p* \Rightarrow_{ss} *mset* (*map* ($\lambda \tau. \text{image-mset } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau)))$)
p) (*ts-list n x*)
| *spp-split*: *mp* = *add-mset* (*add-mset s* (*add-mset t eqc*)) *mp'*
 \Longrightarrow *is-Var s* \neq *is-Var t* \Longrightarrow *eqc* \neq {#} \vee *mp'* \neq {#}
 \Longrightarrow *add-mset mp p* \Rightarrow_{ss} {#*add-mset* {# {# *s*, *t* #} #} *p*, *add-mset* (*add-mset*
(*add-mset s 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>

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>

lemma *spp-step-ms-size*: **assumes** *p* \Rightarrow_{ss} *Q* **and** *q* $\in \# \ Q$
shows *size q* \leq *size p*
<proof>

lemma *spp-step-ms*: **assumes** *p* \Rightarrow_{ss} *Pn*
and *finite-constructor-form-pat* (*mset3 p*)
shows *Ball* (*set-mset Pn*) ($\lambda p'. \text{finite-constructor-form-pat } (\text{mset3 } p')$)

simple-pat-complete C SS ($mset3$ p) \longleftrightarrow $Ball$ (set - $mset$ Pn) ($\lambda p'$. *simple-pat-complete* C SS ($mset3$ p'))
 $\langle proof \rangle$

lemma *finite-tvars-spat-mset3*: *finite* (*tvars-spat* ($mset3$ p))
 $\langle proof \rangle$

lemma *finite-tvars-smp-mset2*: *finite* (*tvars-smp* ($mset2$ mp))
 $\langle proof \rangle$

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
 $\langle proof \rangle$

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 $\exists x \iota. t = Var(x, \iota) \wedge card\text{-of-sort } C \iota \leq size$ p
 $\langle proof \rangle$

definition *max-depth-sort* :: $'s \Rightarrow nat$ **where**

max-depth-sort $s = Maximum$ (*depth-gterm* ' $\{t. t : s \text{ in } \mathcal{T}(C)\}$)

lemma *max-depth-sort-wit*: **assumes** *finite-sort* C s

and $s \in S$
shows $\exists t. t : s \text{ in } \mathcal{T}(C) \wedge$
depth-gterm $t = max\text{-depth-sort}$ $s \wedge$
 $(\forall t'. t' : s \text{ in } \mathcal{T}(C) \longrightarrow \text{depth-gterm } t' \leq \text{depth-gterm } t)$
 $\langle proof \rangle$

lemma *max-depth-sort-Fun*: **assumes** $f: f : \sigma s \rightarrow s \text{ in } C$

and $si: si \in set$ σs
and $fins: finite\text{-sort}$ C s
shows *max-depth-sort* $si < max\text{-depth-sort}$ s
 $\langle proof \rangle$

lemma *max-depth-sort-var*: **assumes** $t : s \text{ in } \mathcal{T}(C, \mathcal{V} \mid SS)$

and $x \in \text{vars } t$
and $\text{finite-sort } C \ s$
shows $\text{max-depth-sort } (\text{snd } x) \leq \text{max-depth-sort } s$
 ⟨*proof*⟩

definition $\text{max-depth-sort-smp} :: ('f, 's) \text{ simple-match-problem-ms} \Rightarrow \text{nat}$ **where**
 $\text{max-depth-sort-smp } mp = \text{Max } (\text{insert } 0 \ (\text{max-depth-sort } ' \text{snd } ' \text{tvars-smp } (\text{mset2 } mp)))$

definition $\text{max-depth-sort-p} :: ('f, 's) \text{ simple-pat-problem-ms} \Rightarrow \text{nat}$ **where**
 $\text{max-depth-sort-p } p = \text{sum-mset } (\text{image-mset } \text{max-depth-sort-smp } p)$

lemma $\text{max-depth-sort-p-add[simp]}$: $\text{max-depth-sort-p } (\text{add-mset } mp \ p) =$
 $\text{max-depth-sort-smp } mp + \text{max-depth-sort-p } p$
 ⟨*proof*⟩

lemma $\text{finite-constructor-form-pat-add}$: $\text{finite-constructor-form-pat } (\text{mset3 } (\text{add-mset } mp \ p))$
 $= (\text{finite-constructor-form-mp } (\text{mset2 } mp) \wedge \text{finite-constructor-form-pat } (\text{mset3 } p))$
 ⟨*proof*⟩

lemma mde-decrease-inst : **assumes** $ft: \text{is-Fun } t$
and $\tau: \tau \in \tau s \ n \ x$
and $mp: mp = \{\#\{\#\text{Var } x, t\#\}\#\}$
and $fin: \text{finite-constructor-form-mp } (\text{mset2 } mp)$
shows $\text{max-depth-sort-smp } mp > \text{max-depth-sort-smp } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau)) \ mp)$
 ⟨*proof*⟩

lemma $\text{mde-weak-decrease-inst}$: **assumes** $\tau: \tau \in \tau s \ n \ x$
and $fin: \text{finite-sort } C \ (\text{snd } x)$
shows $\text{max-depth-sort-smp } mp \geq \text{max-depth-sort-smp } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau)) \ mp)$
 ⟨*proof*⟩

lemma max-depth-sort-le : **assumes** $\text{tvars-smp } (\text{mset2 } mp) \subseteq \text{tvars-smp } (\text{mset2 } mp')$
shows $\text{max-depth-sort-smp } mp \leq \text{max-depth-sort-smp } mp'$
 ⟨*proof*⟩

lemma mp-step-tvars : $mp \rightarrow_{ss} mp' \Longrightarrow \text{tvars-smp } (\text{mset2 } mp') \subseteq \text{tvars-smp } (\text{mset2 } mp)$
 ⟨*proof*⟩

lemma $\text{max-depth-sort-p-inst}$: **assumes** $\text{mem}: \{\#\{\#\text{Var } x, t\#\}\#\} \in \# \ p$
and $t: \text{is-Fun } t$
and $\tau: \tau \in \tau s \ n \ x$
and $fin: \text{finite-constructor-form-pat } (\text{mset3 } p)$

shows $\text{max-depth-sort-p } p > \text{max-depth-sort-p } (\text{image-mset } (\text{image-mset } (\text{image-mset } (\lambda t. t \cdot \tau))) p)$
 ⟨proof⟩

lemma $\text{depth-gterm-le-card: finite-sort } C \sigma \implies t : \sigma \text{ in } \mathcal{T}(D) \implies D \subseteq_m C \implies \text{depth-gterm } t \leq \text{card } (\text{dom } D)$
 ⟨proof⟩

lemma $\text{max-depth-sort-smp-le-card: assumes finite-constructor-form-mp } (mset2 \text{ mp})$
shows $\text{max-depth-sort-smp } mp \leq \text{card } (\text{dom } C)$
 ⟨proof⟩

lemma $\text{max-depth-sort-p-le-card-size: assumes finite-constructor-form-pat } (mset3 \text{ p})$
shows $\text{max-depth-sort-p } p \leq \text{card } (\text{dom } C) * \text{size } p$
 ⟨proof⟩

definition $\text{num-syms-smp} :: ('f, 's)\text{simple-match-problem-ms} \Rightarrow \text{nat}$ **where**
 $\text{num-syms-smp } mp = \text{sum-mset } (\text{image-mset } \text{num-syms } (\text{sum-mset } mp))$

lemma $\text{num-syms-subset: assumes sum-mset } mp \subset\# \text{sum-mset } mp'$
shows $\text{num-syms-smp } mp < \text{num-syms-smp } mp'$
 ⟨proof⟩

definition max-dupl-smp **where**
 $\text{max-dupl-smp } mp = \text{Max } (\text{insert } 0 ((\lambda x. (\sum t \in\# \text{sum-mset } mp. \text{count } (\text{syms-term } t) (\text{Inl } x))) \text{ 'tvars-smp } (mset2 \text{ mp})))$

definition $\text{max-dupl-p} :: ('f, 's)\text{simple-pat-problem-ms} \Rightarrow \text{nat}$ **where**
 $\text{max-dupl-p } p = \text{sum-mset } (\text{image-mset } \text{max-dupl-smp } p)$

lemma $\text{max-dupl-smp: } (\sum t \in\# \text{sum-mset } mp. \text{count } (\text{syms-term } t) (\text{Inl } x)) \leq \text{max-dupl-smp } mp$
 ⟨proof⟩

lemma $\text{max-dupl-smp-le-num-syms: max-dupl-smp } mp \leq \text{num-syms-smp } mp$
 ⟨proof⟩

lemma $\text{num-syms-smp-}\tau\text{s: assumes } \tau : \tau \in \tau\text{s } n \text{ } x$
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⟩

lemma $\text{max-dupl-smp-}\tau\text{s: assumes } \tau : \tau \in \tau\text{s } n \text{ } x$
and $\text{disj: fst 'tvars-smp } (mset2 \text{ mp}) \cap \{n..<n + m\} = \{\}$

shows $\text{max-dupl-smp} (\text{image-mset} (\text{image-mset} (\lambda t. t \cdot \tau)) mp) \leq \text{max-dupl-smp} mp$
 ⟨proof⟩

definition $\text{num-syms-p} :: ('f, 's)\text{simple-pat-problem-ms} \Rightarrow \text{nat}$ **where**
 $\text{num-syms-p } p = \text{sum-mset} (\text{image-mset } \text{num-syms-smp } p)$

lemma $\text{num-syms-p-add[simp]}$: $\text{num-syms-p} (\text{add-mset } mp \ p) = \text{num-syms-smp } mp + \text{num-syms-p } p$
 ⟨proof⟩

lemma $\text{max-dupl-p-le-num-syms}$: $\text{max-dupl-p } p \leq \text{num-syms-p } p$
 ⟨proof⟩

lemma $\text{max-dupl-p-add[simp]}$: $\text{max-dupl-p} (\text{add-mset } mp \ p) = \text{max-dupl-smp } mp + \text{max-dupl-p } p$
 ⟨proof⟩

lemma $\text{max-dupl-mono-main}$: **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 $\text{tvars-smp} (\text{mset2 } mp) \subseteq \text{tvars-smp} (\text{mset2 } mp')$
shows $\text{max-dupl-smp } mp \leq \text{max-dupl-smp } mp'$
 ⟨proof⟩

lemma max-dupl-mono : **assumes** $\text{sum-mset } mp \subseteq \# \text{sum-mset } mp'$
shows $\text{max-dupl-smp } mp \leq \text{max-dupl-smp } mp'$
 ⟨proof⟩

definition $\text{syms-smp} :: ('f, 's)\text{simple-match-problem-ms} \Rightarrow (\text{nat} \times 's + 'f) \text{multiset}$
where
 $\text{syms-smp } mp = \text{sum-mset} (\text{image-mset } \text{syms-term} (\text{sum-mset } mp))$

definition $\text{syms-ecq} :: ('f, \text{nat} \times 's)\text{term multiset} \Rightarrow (\text{nat} \times 's + 'f) \text{multiset}$ **where**
 $\text{syms-ecq } eqc = \text{sum-mset} (\text{image-mset } \text{syms-term } eqc)$

lemma syms-smp-to-ecq : $\text{syms-smp } mp = \text{sum-mset} (\text{image-mset } \text{syms-ecq } mp)$
 ⟨proof⟩

lemma $\text{nums-ec-size-syms-smp}$: $\text{num-syms-smp } mp = \text{size} (\text{syms-smp } mp)$
 ⟨proof⟩

lemma $\text{num-syms-sym-subset}$: **assumes** $\text{syms-smp } mp \subset \# \text{syms-smp } mp'$
shows $\text{num-syms-smp } mp < \text{num-syms-smp } mp'$
 ⟨proof⟩

lemma num-syms-smp-step : $mp \rightarrow_{ss} mp' \implies \text{finite-constructor-form-mp} (\text{mset2}$

mp)
 $\implies \text{num-syms-smp } mp' < \text{num-syms-smp } mp$
 $\langle \text{proof} \rangle$

lemma $\text{num-syms-smp-pos}[simp]$: **assumes** $\text{finite-constructor-form-mp } (mset2\ mp)$

and $mp \neq \{\#\}$
shows $\text{num-syms-smp } mp > 0$
 $\langle \text{proof} \rangle$

lemma count-syms-smp : $(\sum t \in \# \sum \# \text{mp. count } (\text{syms-term } t) (\text{Inl } x)) = (\text{count } (\text{syms-smp } mp) (\text{Inl } x))$
 $\langle \text{proof} \rangle$

lemma max-dupl-smp-step : $mp \rightarrow_{ss} mp' \implies \text{max-dupl-smp } mp' \leq \text{max-dupl-smp } mp$
 $\langle \text{proof} \rangle$

definition $\text{measure-p} :: ('f, 's) \text{simple-pat-problem-ms} \Rightarrow \text{nat}$ **where**
 $\text{measure-p } p = \text{max-depth-sort-p } p * (\text{max-dupl-p } p * m + 1) + \text{num-syms-p } p$

lemma measure-p-bound : **assumes** $\text{finite-constructor-form-pat } (mset3\ p)$
shows $\text{measure-p } p \leq \text{card } (\text{dom } C) * \text{size } p * (\text{num-syms-p } p + 1) * (m + 1)$
 $\langle \text{proof} \rangle$

lemma $\text{measure-p-num-syms}$: **assumes** $\text{max-depth-sort-p } p \leq \text{max-depth-sort-p } p'$

and $\text{max-dupl-p } p \leq \text{max-dupl-p } p'$
and $\text{num-syms-p } p < \text{num-syms-p } p'$
shows $\text{measure-p } p < \text{measure-p } p'$
 $\langle \text{proof} \rangle$

lemma $\text{spp-step-complexity-and-termination}$:

assumes $p \Rightarrow_{ss} Pn$
and $\text{finite-constructor-form-pat } (mset3\ p)$
and $pn \in \# Pn$
shows $\text{measure-p } pn < \text{measure-p } p$
 $\langle \text{proof} \rangle$

$\text{snd} = \text{Simplified Non-Deterministic}$

inductive-set $\text{snd-step} :: ('f, 's) \text{simple-pat-problem-ms} \text{ rel } (\langle \Rightarrow_{\text{snd}} \rangle)$
where $\text{snd-step}: p \Rightarrow_{ss} Q \implies \text{finite-constructor-form-pat } (mset3\ p) \implies q \in \#$
 $Q \implies (p, q) \in \Rightarrow_{\text{snd}}$

lemma snd-step-measure : **assumes** $(p, q) \in \Rightarrow_{\text{snd}}$
shows $\text{measure-p } p > \text{measure-p } q$

<proof>

lemma *snd-bound-steps*: **assumes** $(p,q) \in \Rightarrow_{snd} \widetilde{\sim} n$
shows $measure-p\ q + n \leq measure-p\ p$
<proof>

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>

lemma *SN-snd-step*: $SN \Rightarrow_{snd}$
<proof>

lemma *snd-step-size*: **assumes** $(p,q) \in \Rightarrow_{snd}$
shows $size\ p \geq size\ q$
<proof>

lemma *snd-step-finite-constructor-form-pat*: **assumes** $(p,q) \in \Rightarrow_{snd}$
shows *finite-constructor-form-pat* (*mset3* q)
<proof>

lemma *snd-step-completeness*: **assumes** $p \notin NF \Rightarrow_{snd}$
shows *simple-pat-complete* $C\ SS\ (mset3\ p) \longleftrightarrow (\forall\ q. (p,q) \in \Rightarrow_{snd} \longrightarrow$
simple-pat-complete $C\ SS\ (mset3\ q))$
(is *?left* = *?right*)
<proof>

lemma *snd-steps*: **assumes** $(p,q) \in (\Rightarrow_{snd})^*$
shows $size\ p \geq size\ q$
simple-pat-complete $C\ SS\ (mset3\ p) \Longrightarrow$ *simple-pat-complete* $C\ SS\ (mset3\ q)$
finite-constructor-form-pat (*mset3* p) \Longrightarrow *finite-constructor-form-pat* (*mset3* q)
<proof>

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* $C\ SS\ (mset3\ q)$
<proof>

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* $C\ SS\ (mset3\ q))$
<proof>

lemma *snd-steps-NF-fvf-small-sort*: **assumes** *finite-constructor-form-pat* (*mset3* p)
and $(p,q) \in \Rightarrow_{snd}^!$
shows $\forall\ mp \in \# q. \forall\ eqc \in \# mp. \forall\ t \in \# eqc. \exists\ x\ \iota. t = Var\ (x,\ \iota) \wedge card-of-sort$
 $C\ \iota \leq size\ p$
<proof>

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)*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)*simple-pat-problem-ms* multiset \Rightarrow bool **where**
spp-det-prob $P = (\forall p \in \# P. \text{finite-constructor-form-pat } (mset3\ p))$

definition *spp-pat-complete* :: ('f,'s)*simple-pat-problem-ms* multiset \Rightarrow 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* $(\text{add-mset } p\ P) =$
 $(\text{finite-constructor-form-pat } (mset3\ p) \wedge \text{spp-det-prob } P)$
 $\langle \text{proof} \rangle$

lemma *spp-det-prob-plus[simp]*: *spp-det-prob* $(P + P') =$
 $(\text{spp-det-prob } P \wedge \text{spp-det-prob } P')$
 $\langle \text{proof} \rangle$

lemma *spp-det-prob-empty[simp]*: *spp-det-prob* $\{\#\}$
 $\langle \text{proof} \rangle$

lemma *spp-det-step-ms*: $(P, P') \in \Rightarrow_{ss} \Longrightarrow$
spp-pat-complete $P = \text{spp-pat-complete } P' \wedge (\text{spp-det-prob } P \longrightarrow \text{spp-det-prob } P')$
 $\langle \text{proof} \rangle$

lemma *spp-det-steps-ms*: $(P, P') \in \Rightarrow_{ss}^* \Longrightarrow$
spp-pat-complete $P = \text{spp-pat-complete } P' \wedge (\text{spp-det-prob } P \longrightarrow \text{spp-det-prob } P')$
 $\langle \text{proof} \rangle$

lemma *SN-spp-det-step*: $SN \Rightarrow_{ss}$
 ⟨proof⟩

lemma *NF-spp-det-step*: **assumes** *spp-det-prob* P
and $P \in NF \Rightarrow_{ss}$
shows $P = \{\#\} \vee P = \{\#\{\#\}\#$
 ⟨proof⟩

lemma *decision-procedure-spp-det*:
assumes *valid-input*: *spp-det-prob* P **and** *NF*: $(P, Q) \in \Rightarrow_{ss}!$
shows $Q = \{\#\} \wedge$ *spp-pat-complete* P — either the result is and input P is complete
 $\vee Q = \{\#\{\#\}\# \wedge \neg$ *spp-pat-complete* P — or the result = bot and P is not complete
 ⟨proof⟩

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* \Rightarrow (f, s) *simple-match-problem-ms*
where
 $fcf-mp-to-smp\ mp = (let\ xs = mset-set\ ((the-Var\ o\ snd)\ 'set-mset\ mp)$
 $in\ image-mset\ (\lambda\ x.\ image-mset\ fst\ (filter-mset\ (\lambda\ (t, l).\ l = Var\ x)\ mp))\ xs)$

lemma *fcf-mp-to-smp*: **assumes** *fcf*: *finite-constr-form-mp* C (*set-mset* mp)
and *no-fail*: \neg *match-fail* mp
and *wf-match*: *wf-match* (*set-mset* mp)
shows *finite-constructor-form-mp* (*mset2* (*fcf-mp-to-smp* mp))
 ⟨proof⟩

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))
 ⟨proof⟩

lemma *wf-pat-nd-step*: $(p, q) \in \Rightarrow_{nd} \Longrightarrow$ *wf-pat* (*pat-mset* p) \Longrightarrow *wf-pat* (*pat-mset* q)
 ⟨proof⟩

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

<proof>

lemma *num-syms-smp-fcf-mp-to-smp-is-meas-tsymbols:*
 assumes *finite-constr-form-mp C (mp-mset mp)*
 shows *num-syms-smp (fcf-mp-to-smp mp) = num-tsyms-mp mp*
<proof>

lemma *num-syms-p-fcf-pat-to-spat-is-meas-tsymbols:*
 assumes *finite-constr-form-pat C (pat-mset p)*
 shows *num-syms-p (fcf-pat-to-spat p) = meas-tsymbols p*
<proof>

lemma *measure-pat-poly-to-measure-p:*
 assumes *finite-constr-form-pat C (pat-mset p)*
 and *finite-constructor-form-pat (mset3 (fcf-pat-to-spat p))*
 and *c: card (dom C) * size p ≤ c*
 shows *measure-p (fcf-pat-to-spat p) ≤ measure-pat-poly c p*
<proof>

theorem *complexity-combined: fixes p :: ('f,'v,'s)pat-problem-mset*
 assumes *impr: improved infinite (UNIV :: 'v set)*
 and *wf: wf-pat (pat-mset p)*
 and *phase1: (p,p1) ∈ ⇒_{nd} $\overset{\sim}{\sim}$ n1 (p,p1) ∈ ⇒_{nd}[!]*
 and *translation: p1' = fcf-pat-to-spat p1*
 and *phase2: (p1',p2) ∈ ⇒_{snd} $\overset{\sim}{\sim}$ n2*
 shows *n1 + n2 + num-syms-p p2 ≤ (card (dom C) * size p + num-syms-pat p) * (num-syms-pat p * m + 2)*
 size p2 ≤ size p
 p2 ∈ NF ⇒_{snd} ⇒ mp ∈# p2 ⇒ eqc ∈# mp ⇒ t ∈# eqc ⇒ (x,t) ∈ vars
 t ⇒ card-of-sort C ι ≤ size p
<proof>
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 *fuf: finite-var-form-pat C (pat-list pp)*
 and *pp: pp = pat-of-var-form-list fuf*
 and *dist: Ball (set fuf) (distinct o map fst)*

shows *pat-complete* C (*pat-list* pp) \longleftrightarrow
 $(\forall \alpha. (\forall v \in \text{set } (tvars\text{-}pat\text{-}list\ pp). \alpha\ v < \text{card-of-sort } C\ (snd\ v)) \longrightarrow$
 $(\exists c \in \text{set } (map\ (map\ snd)\ fvf).$
 $\forall vs \in \text{set } c. \text{UNIQ } (\alpha\ ' \text{set } vs)))$
 $\langle proof \rangle$

lemma *pat-complete-via-cnf*:

assumes *fvf*: *finite-var-form-pat* C (*pat-list* pp)
and *pp*: $pp = \text{pat-of-var-form-list } fvf$
and *dist*: $\text{Ball } (\text{set } fvf) (\text{distinct } o\ \text{map } fst)$
and *cnf*: $cnf = \text{map } (map\ snd)\ fvf$
shows *pat-complete* C (*pat-list* pp) \longleftrightarrow
 $(\forall \alpha. (\forall v \in \text{set } (\text{concat } (\text{concat } cnf))). \alpha\ v < \text{card-of-sort } C\ (snd\ v)) \longrightarrow$
 $(\exists c \in \text{set } cnf. \forall vs \in \text{set } c. \text{UNIQ } (\alpha\ ' \text{set } vs)))$
 $\langle proof \rangle$

fun *zip-lists* :: $\text{nat} \Rightarrow 'a\ list\ list \Rightarrow 'a\ list\ list$ **where**

$zip\text{-lists } n\ [] = \text{replicate } n\ []$
 $| zip\text{-lists } n\ (xs\ \# \ xss) = \text{map2 } (\#)\ xs\ (zip\text{-lists } n\ xss)$

lemma *zip-lists*: **assumes** $\bigwedge xs. xs \in \text{set } xss \implies \text{length } xs = n$
shows $zip\text{-lists } n\ xss = \text{map } (\lambda i. \text{map } (\lambda xs. xs\ !\ i)\ xss)\ [0..<n]$
 $\langle proof \rangle$

fun *length-gt-1* **where**

$\text{length-gt-1 } (x\ \# \ y\ \# \ xs) = \text{True}$
 $| \text{length-gt-1 } - = \text{False}$

lemma *length-gt-1[simp]*: $\text{length-gt-1 } xs = (\text{length } xs > 1)$
 $\langle proof \rangle$

definition *uniq-list* :: $'a\ list \Rightarrow \text{bool}$ **where**

$\text{uniq-list } xs = (xs = [] \vee \text{is-singleton-list } xs)$

lemma *uniq-list[simp]*: $\text{uniq-list } xs \longleftrightarrow \text{UNIQ } (\text{set } xs)$
 $\langle proof \rangle$

primrec *extract-option* :: $('a \Rightarrow 'b\ \text{option}) \Rightarrow 'a\ list \Rightarrow ('a\ list \times 'a \times 'b \times 'a\ list)\ \text{option}$ **where**

$\text{extract-option } f\ [] = \text{None}$
 $| \text{extract-option } f\ (x\ \# \ xs) = (\text{case } f\ x\ \text{of } \text{Some } y \Rightarrow \text{Some } ([], x, y, xs)$
 $| \text{None} \Rightarrow \text{map-option } (\text{map-prod } (\text{Cons } x)\ \text{id})\ (\text{extract-option } f\ xs))$

lemma *extract-option-Some*: **assumes** $\text{extract-option } f\ xs = \text{Some } (\text{bef}, x, y, \text{aft})$

shows $xs = \text{bef } @\ x\ \# \ \text{aft } f\ x = \text{Some } y$
 $\langle proof \rangle$

lemma *extract-option-None*: $\text{extract-option } f \text{ } xs = \text{None} \iff \text{Ball } (\text{set } xs) (\lambda x. f \text{ } x = \text{None})$
 ⟨proof⟩

fun *sort-of* :: $('f \times 's, 'v \times 's)\text{term} \Rightarrow 's$ **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⟩
 ⟨proof⟩

fun *list-Union* :: 'a set list \Rightarrow 'a set **where**
list-Union [] = {}
 | *list-Union* (x # xs) = x \cup *list-Union* xs

lemma *list-Union*[simp]: $\text{list-Union } xs = \bigcup (\text{set } xs)$
 ⟨proof⟩

fun *sorts-of* :: $('f \times 's, 'v \times 's)\text{term} \Rightarrow 's$ 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* :: $('f \times 's, 'v \times 's)\text{term} \Rightarrow 's$ set **where**
arg-sorts-of t = *list-Union* (map *sorts-of* (args t))

fun *remove-sort* :: $('f \times 's, 'v)\text{term} \Rightarrow ('f, 'v)\text{term}$ **where**
remove-sort (Var x) = Var x
 | *remove-sort* (Fun (f,s) ts) = Fun f (map *remove-sort* ts)

type-synonym $('f, 's)\text{simple-match-problem-list} = ('f, \text{nat} \times 's)\text{term list list}$
type-synonym $('f, 's)\text{simple-match-problem-slist} = ('f \times 's, \text{nat} \times 's)\text{term list list}$
type-synonym $('f, 's)\text{simple-pat-problem-slist} = ('f \times 's, \text{nat} \times 's)\text{term list list list}$
type-synonym $('f, 's)\text{tagged-simple-pat-problem-slist} = \text{bool} \times ('f, 's)\text{simple-pat-problem-slist}$

definition *search-fun-pp* :: $('f, 's)\text{simple-pat-problem-slist} \Rightarrow -$ 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 $t \in \bigcup (\bigcup (\text{set3 } pp)) \implies \text{is-Var } t$
 ⟨proof⟩

lemma *search-fun-pp-Some*: **assumes** *search-fun-pp* pp = Some (p1, mp, (mp1, eqc, (eqc1, t, eqc2), mp2), p2)
shows $pp = p1 @ mp \# p2 \text{ } mp = mp1 @ eqc \# mp2 \text{ } eqc = eqc1 @ t \# eqc2$
is-Fun t

<proof>

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*
<proof>

lemma *remove-add-sort[simp]*: *remove-sort (add-sort t)* = *t*
<proof>

lemma *is-Var-add-sort[simp]*: *is-Var (add-sort t)* = *is-Var t*
<proof>

lemma *inj-add-sort[simp]*: *inj add-sort*
<proof>

lemma *add-sort-inj[simp]*: (*add-sort s* = *add-sort t*) = (*s* = *t*)
<proof>

lemma *add-sort*: **assumes** *t : s in T(C, V)*
shows *sort-of (add-sort t)* = *s*
<proof>

definition *rel-term* :: (*f*,*nat* × *s*)*term* ⇒ (*f* × *s*,*nat* × *s*) *term* ⇒ *bool* **where**
rel-term t st = (*st* = *add-sort t*)

definition *rel-smp* :: (*f*,*s*)*simple-match-problem-list*
⇒ (*f*,*s*)*simple-match-problem-slist*
⇒ (*f*,*s*)*simple-match-problem-ms* ⇒ *bool* **where**
rel-smp mpl mpsl mpm = (*finite-constructor-form-mp (set2 mpl)*
∧ *mpsl* = *map (map add-sort) mpl* ∧ *mpm* = *mset (map mset mpl)*)

abbreviation *mset2'* **where** *mset2' mpl* ≡ *mset (map mset mpl)*

abbreviation *mset3'* **where** *mset3' ppl* ≡ *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
⟨proof⟩

```

definition *large-sort-impl-main* :: ('f,'s)simple-pat-problem-slist ⇒ 's ⇒ ('f,'s)simple-pat-problem-slist option

```

where large-sort-impl-main p s = (let
  find-conflict = (λ mp. (∃ eqc ∈ set mp. sort-of (hd eqc) = s))
  in case partition find-conflict p of
    (del, keep) ⇒ if del ≠ [] ∧ length del < cd-sort s ∧
      (∀ mp ∈ set keep. ∀ eq ∈ set mp. ∀ t ∈ set eq. ∀ x ∈ set (vars-term-list
t). snd x ≠ s)
      then Some keep
      else None)

```

definition *large-sort-impl* :: ('f,'s)simple-pat-problem-slist ⇒ ('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 (λ (-,-,p',-). p') (extract-option (large-sort-impl-main p) sorts))

```

function *simplify-mp-main* :: ('f,'s)simple-match-problem-slist ⇒ ('f,'s)simple-match-problem-slist ⇒ ('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 ∨ 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 ∈ set roots
      then (if Ball (set eqc') (λ t. Ball (set eqc') (λ s. ¬ 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) (λ eqc. uniq-list (map sort-of eqc)) then
simplify-mp-main (new-eqcs @ mp) mpout else None)
        else None)))
⟨proof⟩

```

termination

⟨proof⟩

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
 $\text{simplify-pp } [] = \text{Some } []$
 $| \text{simplify-pp } (mp \# p) = (\text{case } \text{simplify-mp } mp \text{ of}$
 $\quad \text{None} \Rightarrow \text{simplify-pp } p$
 $\quad | \text{Some } mp' \Rightarrow \text{if } mp' = [] \text{ then None else map-option } (\text{Cons } mp') (\text{simplify-pp } p))$

fun $\text{simplify-tpp} :: ('f, 's)\text{tagged-simple-pat-problem-slist} \Rightarrow ('f, 's)\text{tagged-simple-pat-problem-slist}$
option where
 $\text{simplify-tpp } (\text{True}, p) = \text{Some } (\text{True}, p)$
 $| \text{simplify-tpp } (\text{False}, p) = \text{map-option } (\text{Pair } \text{True}) (\text{simplify-pp } p)$

definition $\text{s}\tau\text{c} :: \text{nat} \Rightarrow \text{nat} \times 's \Rightarrow 'f \times 's \text{ list} \Rightarrow ('f \times 's, \text{nat} \times 's)\text{subst}$ **where**
 $\text{s}\tau\text{c } n \ x = (\lambda(f, ss). \text{subst } x \ (\text{Fun } (f, \text{snd } x) \ (\text{map } \text{Var } (\text{indexed-from } n \ ss))))$

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

lemma $\text{add-sort-}\tau\text{c}$: **assumes** $f : ss \rightarrow \text{snd } x \text{ in } C$
 $t : \iota \text{ in } \mathcal{T}(C, \mathcal{V})$
shows $\text{add-sort } (t \cdot \text{s}\tau\text{c } n \ x \ (f, ss)) = \text{add-sort } t \cdot \text{s}\tau\text{c } n \ x \ (f, ss)$
<proof>

definition $\text{inst-list} :: \text{nat} \Rightarrow \text{nat} \times 's \Rightarrow ('f, 's)\text{simple-pat-problem-slist} \Rightarrow ('f, 's)\text{simple-pat-problem-slist}$
list
where $\text{inst-list } n \ x \ p = \text{map } (\lambda \tau. \text{map } (\text{map } (\text{map } (\lambda t. t \cdot \tau))) \ p) \ (\text{s}\tau\text{s-list } n \ x)$

definition $\text{full-step} :: \text{nat} \Rightarrow ('f, 's)\text{tagged-simple-pat-problem-slist} \Rightarrow ('f, 's)\text{tagged-simple-pat-problem-slist}$
list + (nat × 's) list list list where
 $\text{full-step } n \ p = (\text{case } \text{simplify-tpp } p \text{ of None} \Rightarrow \text{Inl } []$
 $\quad | \text{Some } (\text{True}, p') \Rightarrow (\text{case } \text{large-sort-impl } p' \text{ of}$
 $\quad \quad \text{Some } p2 \Rightarrow (\text{Inl } [(\text{True}, p2)])$
 $\quad \quad | \text{None} \Rightarrow (\text{case } \text{search-fun-pp } p' \text{ of}$
 $\quad \quad \quad \text{None} \Rightarrow \text{Inr } (\text{map } (\text{map } (\text{map } \text{the-Var})) \ p')$
 $\quad \quad \quad | \text{Some } (p1, mp, (mp1, eqc, (eqc1, t, eqc2), mp2), p2) \Rightarrow$
 $\quad \quad \quad \text{let } x = \text{the } (\text{find is-Var } eqc)$
 $\quad \quad \quad \text{in if } mp = [[t, x]] \vee mp = [[x, t]] \text{ then Inl } (\text{map } (\text{Pair } \text{False}) \ (\text{inst-list } n \ (\text{the-Var } x) \ p'))$
 $\quad \quad \quad \text{else let } eqn = eqc1 \ @ \ eqc2; mpn = (\text{if } \text{length-gt-1 } eqn \ \text{then } eqn \ # \ mp1 \ @ \ mp2 \ \text{else } mp1 \ @ \ mp2)$
 $\quad \quad \quad \text{in Inl } ((\text{True}, mpn \ # \ p1 \ @ \ p2)$
 $\quad \quad \quad \# \ \text{map } (\text{Pair } \text{False}) \ (\text{inst-list } n \ (\text{the-Var } x) \ ([[x, t]] \ # \ p1 \ @ \ p2))))))$

definition $\text{fidl-encoder} :: ('x \times 's) \text{ list list list} \Rightarrow ('x, 's) \text{ fidl-input}$ **where**

$fidl\text{-}encoder\ p = (let\ vars = remdups\ (concat\ (concat\ p))$
 $in\ (map\ (\lambda\ x.\ (x,\ int\ (cd\text{-}sort\ (snd\ x))))\ vars,\ map\ (List.maps\ (\lambda\ eqc.\ zip\ eqc$
 $(tl\ eqc)))\ p))$

context

fixes $fidl\text{-}solver :: (nat, 's)\ fidl\text{-}input \Rightarrow bool$

begin

definition $fvf\text{-}solver\ fvf = (\neg\ fidl\text{-}solver\ (bounds\text{-}list\ (cd\text{-}sort\ \circ\ snd)\ fvf,\ dist\text{-}pairs\text{-}list$
 $fvf))$

partial-function (*tailrec*) $fcf\text{-}solver\text{-}loop$ **where**

$fcf\text{-}solver\text{-}loop\ n\ P = (case\ P\ of\ [] \Rightarrow True$
 $| p\ \# ps \Rightarrow (case\ full\text{-}step\ n\ p\ of$
 $Inl\ ps1 \Rightarrow fcf\text{-}solver\text{-}loop\ (n + m)\ (ps1\ @\ ps)$
 $| Inr\ fvf \Rightarrow if\ fvf\text{-}solver\ fvf\ then\ fcf\text{-}solver\text{-}loop\ n\ ps$
 $else\ False))$

abbreviation $add1$ **where** $add1 \equiv map\ add\text{-}sort$

abbreviation $add2$ **where** $add2 \equiv map\ add1$

abbreviation $add3$ **where** $add3 \equiv map\ add2$

definition $fcf\text{-}solver\text{-}alg\ n\ p = fcf\text{-}solver\text{-}loop\ n\ [(False,\ add3\ p)]$
end

lemma $mset2\text{-}mset2'\text{-}set2[simp]$: $mset2\ (mset2'\ mp) = set2\ mp$
 $\langle proof \rangle$

lemma $mset3\text{-}mset3'\text{-}set3[simp]$: $mset3\ (mset3'\ p) = set3\ p$
 $\langle proof \rangle$

end

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

begin

lemma $mp\text{-}steps\text{-}cong$: **assumes** *finite-constructor-form-pat* ($mset3\ (add\text{-}mset\ mp$
 $p)$)

shows $(\rightarrow_{ss})^{**}\ mp\ mp' \Longrightarrow$
 $(add\text{-}mset\ (add\text{-}mset\ mp\ p)\ P,\ add\text{-}mset\ (add\text{-}mset\ mp'\ p)\ P) \in (\equiv_{ss})^*$
 $\wedge\ finite\text{-}constructor\text{-}form\text{-}pat\ (mset3\ (add\text{-}mset\ mp'\ p))$

$\langle proof \rangle$

definition $simplified\text{-}mp\ mp = (Ball\ (set\ mp)\ (\lambda\ eqc.\ length\ eqc > 1 \wedge (\exists\ t \in set$
 $eqc.\ is\text{-}Var\ t) \wedge\ distinct\ eqc))$

definition $simplified\text{-}pp\ p = (Ball\ (set\ p)\ (\lambda\ mp.\ simplified\text{-}mp\ mp \wedge\ mp \neq []))$

lemma $simplify\text{-}mp\text{-}main$: **assumes** *rel-smp* $mpl\ (mpsl\ @\ mpsout)\ mpm$

and $res = simplify\text{-}mp\text{-}main\ mpsl\ mpsout$
and $Ball\ (set\ mpsout)\ prop$
and $tvars\text{-}smp\ (set2\ mpl) \subseteq V$
and $prop = (\lambda\ eqc.\ length\ eqc > 1 \wedge (\exists\ t \in\ set\ eqc.\ is\text{-}Var\ t) \wedge\ distinct\ eqc)$
shows $res = Some\ mpsl' \implies \exists\ mpl'\ mpm'.$ $rel\text{-}smp\ mpl'\ mpsl'\ mpm' \wedge (\rightarrow_{ss})^{**}$
 $mpm\ mpm'$
 $\wedge\ Ball\ (set\ mpsl')\ prop \wedge\ tvar\text{-}smp\ (set2\ mpl') \subseteq V$
 $res = None \implies \exists\ mpm'.$ $(\rightarrow_{ss})^{**}\ mpm\ mpm' \wedge\ smp\text{-}fail\text{-}ms\ mpm'$
 $\langle proof \rangle$

lemma $sorts\text{-}of\text{-}subterm:$ $t \supseteq a \implies a : s\ in\ \mathcal{T}(C, \mathcal{V}) \implies s \in\ sorts\text{-}of\ (add\text{-}sort\ t)$
 $\langle proof \rangle$

lemma $vars\text{-}term\text{-}add\text{-}sort[simp]:$ $vars\text{-}term\ (add\text{-}sort\ t) = vars\text{-}term\ t$
 $\langle proof \rangle$

lemma $eroot\text{-}add\text{-}sort:$ **assumes** $t : \iota\ in\ \mathcal{T}(C, \mathcal{V})$
shows $eroot\ (add\text{-}sort\ t) = map\text{-}sum\ (map\text{-}prod\ (\lambda\ f.\ (f, \iota))\ id)\ id\ (eroot\ t)$
 $\langle proof \rangle$

lemma $large\text{-}sort\text{-}impl:$ **assumes** $large\text{-}sort\text{-}impl\ (add3\ p) = Some\ ap'$
and $fin:$ $finite\text{-}constructor\text{-}form\text{-}pat\ (set3\ p)$
and $simpl:$ $simplified\text{-}pp\ p$
shows $\exists\ p'.$ $mset3'\ p \Rightarrow_{ss} \{\# mset3'\ p'\ \#\} \wedge ap' = add3\ p' \wedge set\ p' \subseteq set\ p \wedge$
 $length\ p' \leq length\ p$
 $\langle proof \rangle$

lemma $simplify\text{-}mp:$ **assumes** $finite\text{-}constructor\text{-}form\text{-}mp\ (set2\ mp)$
and $res:$ $res = simplify\text{-}mp\ (add2\ mp)$
and $vars:$ $tvars\text{-}smp\ (set2\ mp) \subseteq V$
shows $res = Some\ amp' \implies$
 $\exists\ mp'.$ $amp' = add2\ mp' \wedge (\rightarrow_{ss})^{**}\ (mset2'\ mp)\ (mset2'\ mp') \wedge simplified\text{-}mp$
 $amp' \wedge tvar\text{-}smp\ (set2\ mp') \subseteq V$
 $res = None \implies \exists\ mp'.$ $(\rightarrow_{ss})^{**}\ (mset2'\ mp)\ mp' \wedge smp\text{-}fail\text{-}ms\ mp'$
 $\langle proof \rangle$

lemma $simplify\text{-}pp:$ **assumes** $finite\text{-}constructor\text{-}form\text{-}pat\ (set3\ p)$
and $res = simplify\text{-}pp\ (add3\ p)$
and $tvars\text{-}spat\ (set3\ p) \subseteq V \wedge length\ p < k$
shows $res = Some\ ap' \implies \exists\ p'.$ $ap' = add3\ p'$
 $\wedge\ (add\text{-}mset\ (mset3'\ p)\ P,\ add\text{-}mset\ (mset3'\ p')\ P) \in (\Rightarrow_{ss})^*$
 $\wedge\ simplified\text{-}pp\ ap'$
 $\wedge\ finite\text{-}constructor\text{-}form\text{-}pat\ (set3\ p')$
 $\wedge\ tvar\text{-}spat\ (set3\ p') \subseteq V \wedge length\ p' < k$
 $(is\ ?A \implies ?B)$
and $res = None \implies (add\text{-}mset\ (mset3'\ p)\ P,\ P) \in (\Rightarrow_{ss})^* (is\ ?C \implies ?D)$
 $\langle proof \rangle$

lemma $inst\text{-}list\text{-}result:$ **assumes** $finite\text{-}constructor\text{-}form\text{-}pat\ (set3\ p)$

shows $inst\text{-}list\ n\ x\ (add3\ p)$
 $=\ map\ add3\ (map\ (\lambda\ \tau.\ (map\ (map\ (map\ (\lambda\ t.\ t\ \cdot\ \tau)))\ p))\ (\tau s\text{-}list\ n\ x))$
 $\langle proof \rangle$

lemma $inst\text{-}list$: **assumes** $\{\#\{\# Var\ x,\ t\#\}\#\} \in\#\ mset3'\ p$
 $is\text{-}Fun\ t$
 $tvars\text{-}spat\ (set3\ p) \subseteq \{..\lt n\} \times UNIV \wedge length\ p < k$
 $finite\text{-}constructor\text{-}form\text{-}pat\ (set3\ p)$
shows $\exists\ ps'.\ mset3'\ p \Rightarrow_{ss}\ mset\ (map\ mset3'\ ps') \wedge map\ add3\ ps' = inst\text{-}list\ n\ x$
 $(add3\ p)$
 $\wedge Ball\ (set\ ps')\ (\lambda\ p'.\ tvars\text{-}spat\ (set3\ p') \subseteq \{..\lt n+m\} \times UNIV \wedge length\ p' <$
 $k)$
 $\langle proof \rangle$

lemma $simplified\text{-}mp\text{-}add2$: $simplified\text{-}mp\ (add2\ mp) = simplified\text{-}mp\ mp$
 $\langle proof \rangle$

lemma $simplified\text{-}pp\text{-}add3$: $simplified\text{-}pp\ (add3\ p) = simplified\text{-}pp\ p$
 $\langle proof \rangle$

fun $simpl\text{-}tag :: ('f,'s)tagged\text{-}simple\text{-}pat\text{-}problem\text{-}slist \Rightarrow bool$ **where**
 $simpl\text{-}tag\ (False,p) = True$
 $| simpl\text{-}tag\ (True,p) = simplified\text{-}pp\ p$

lemma $simplify\text{-}tpp$: **assumes** inv : $simpl\text{-}tag\ (tag,\ add3\ p)$
and res : $res = simplify\text{-}tpp\ (tag,\ add3\ p)$
and fin : $finite\text{-}constructor\text{-}form\text{-}pat\ (set3\ p)$
and $vars$: $tvars\text{-}spat\ (set3\ p) \subseteq V \wedge length\ p < k$
shows $res = Some\ (tag',ap') \Longrightarrow \exists\ p'.\ ap' = add3\ p'$
 $\wedge (add\text{-}mset\ (mset3'\ p)\ P,\ add\text{-}mset\ (mset3'\ p')\ P) \in (\Rightarrow_{ss})^*$
 $\wedge tag' = True$
 $\wedge simpl\text{-}tag\ (tag',ap')$
 $\wedge finite\text{-}constructor\text{-}form\text{-}pat\ (set3\ p')$
 $\wedge tvars\text{-}spat\ (set3\ p') \subseteq V \wedge length\ p' < k$
and $res = None \Longrightarrow (add\text{-}mset\ (mset3'\ p)\ P,\ P) \in (\Rightarrow_{ss})^*$
 $\langle proof \rangle$

abbreviation $add4$ **where** $add4 \equiv map\ (map\text{-}prod\ id\ add3)$

abbreviation $mset4'$ **where** $mset4' \equiv image\text{-}mset\ (mset3'\ o\ snd)\ o\ mset$

lemma $full\text{-}step$: **assumes** $tvarsp$: $tvars\text{-}spat\ (set3\ p) \subseteq \{..\lt n\} \times UNIV \wedge length$
 $p < k$
and $finp$: $finite\text{-}constructor\text{-}form\text{-}pat\ (set3\ p)$
and $inv\text{-}tag$: $simpl\text{-}tag\ (tag,\ add3\ p)$
and $result$: $full\text{-}step\ n\ (tag,\ add3\ p) = res$
shows $res = Inl\ aps \Longrightarrow \exists\ ps.\ aps = add4\ ps$
 $\wedge (add\text{-}mset\ (mset3'\ p)\ P,\ mset4'\ ps + P) \in (\Rightarrow_{ss})^+$
 $\wedge Ball\ (snd\ ' set\ ps)\ (\lambda\ p'.\ tvars\text{-}spat\ (set3\ p') \subseteq \{..\lt n + m\} \times UNIV$

\wedge length $p' < k$
 \wedge Ball (set aps) simpl-tag
 $res = \text{Inr } fvf \implies \text{simplified-pp } (\text{map } (\text{map } (\text{map } (\text{Var} :: - \implies ('f,-)\text{term}))) fvf) \wedge$
length $fvf < k$
 \wedge (add-mset (mset3' p) P, add-mset (mset3' (map (map (map Var)) fvf)) P)
 $\in (\implies_{ss})^*$
 $\wedge (\forall x \iota. (x, \iota) \in \text{set } (\text{concat } (\text{concat } fvf)) \longrightarrow \text{card-of-sort } C \iota < k)$
<proof>

context

fixes solver :: ((nat × 's) × int)list × - \implies 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: $(\forall x \iota. (x, \iota) \in \text{set } (\text{concat } (\text{concat } cnf)) \longrightarrow \text{card-of-sort } C \iota < k)$

and cnf: cnf = map (map snd) fvf

shows pat-complete C (pat-list pp) $\longleftrightarrow \neg$ solver (bounds-list (cd-sort o snd) cnf, dist-pairs-list cnf)

<proof>

lemma fvf-solver: **assumes** tfvf: tfvf = map (map (map (Var :: nat × 's \implies ('f, nat × 's)term))) fvf

and small: $(\forall x \iota. (x, \iota) \in \text{set } (\text{concat } (\text{concat } fvf)) \longrightarrow \text{card-of-sort } C \iota < k)$

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>

lemma fcf-solver-loop: **assumes** vars: Ball (snd ' set P) $(\lambda p. \text{tvars-spat } (\text{set3 } p) \subseteq \{..<n\} \times \text{UNIV} \wedge \text{length } p < k)$

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)

<proof>

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>

lemma fcf-solver-alg': fcf-solver k (fcf-solver-alg solver)

<proof>

end
end

context *pattern-completeness-context*
begin

lemmas *fcf-solver-code-eqns* =
fcf-solver-alg-def
fcf-solver-loop.simps
fvf-solver-def
full-step-def
simplify-tpg.simps
simplify-pp.simps
simplify-mp-def
simplify-mp-main.simps
add-sort.simps
inst-list-def[*unfolded* *s τ s-list-def* *s τ 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* \Rightarrow (*'v,'s*)*fidl-input option* **where**
delete-trivial-sorts = (λ (*bnds, diffs*).
case partition (($=$) 0 *o snd*) *bnds of*
 ($\square, -$) \Rightarrow *Some (bnds, diffs)*
 | (*triv, non-triv*) \Rightarrow *let* *triv-sorts* = *set (map (snd o fst) triv)*;
 newdiffs = *map (filter (lambda vw. snd (fst vw) \notin *triv-sorts*)) diffs*
 in if $\square \in$ *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* \longrightarrow \neg *fidl-solvable input*) \wedge (*ooutput = Some output* \longrightarrow

fidl-input output \wedge *fidl-solvable input* = *fidl-solvable output*
 <proof>

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 \times 's) \times int) list **and** s :: 's
assumes *filt*: *filt* = *filter* ((=) s \circ *snd* \circ *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* (λ i. (*fst* (*fst* (*filt* ! i)), *snd* (*hd* *filt*) - int i) [0..*length* *filt*]))))
 <proof>

definition *find-large-sorts* :: (('v \times 's) \times int) list \Rightarrow 's set **where**
find-large-sorts bnds = (let
 m = *assign-by-sort* *Mapping.empty* bnds;
 mf = *Mapping.filter* (λ 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)*fidl-input* \Rightarrow ('v,'s)*fidl-input* \times bool
where
delete-large-sorts-single = (λ (bnds, *diffs*).
 let *lsorts* = *find-large-sorts* bnds
 in if *Set.is-empty* *lsorts* then ((bnds,*diffs*), *False*)
 else let *newdiffs* = *filter* (λ vs. \forall vw \in set vs. *snd* (*fst* vw) \notin *lsorts*) *diffs*;
 remvars = set (*List.maps* (*List.maps* (λ (v,w). [v,w])) *newdiffs*);
 newbnds = *filter* (λ vb. *fst* vb \in *remvars*) bnds
 in ((*newbnds*, *newdiffs*), *True*))

lemma *delete-large-sorts-single*: **assumes** *inp*: *fidl-input* (*input* :: ('v,'s)*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>

partial-function (*tailrec*) *delete-large-sorts* :: ('v,'s)*fidl-input* \Rightarrow ('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 fdl-input inp and del: delete-large-sorts inp = out shows fdl-input out ∧ fdl-solvable inp = fdl-solvable out*
 ⟨proof⟩

definition *fdl-pre-processor where fdl-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 *fdl-pre-processor: assumes finite-idl-solver solver shows finite-idl-solver (fdl-pre-processor solver)*
 ⟨proof⟩

datatype *'v fdl-constraint = Var-Int (fidlc-vi: 'v × int) | Var-Var (fidlc-vv: 'v × 'v)*

fun *is-fidlc-vi where is-fidlc-vi (Var-Int -) = True | is-fidlc-vi - = False*

fun *fidlc-sat :: ('v ⇒ int) ⇒ 'v fdl-constraint ⇒ bool where fidlc-sat α (Var-Int (v,i)) = (α v ≠ i) | fidlc-sat α (Var-Var (v,w)) = (α v ≠ α w)*

fun *fidlc-vars :: 'v fdl-constraint ⇒ 'v list where fidlc-vars (Var-Int (v,i)) = [v] | fidlc-vars (Var-Var (v,w)) = [v,w]*

lemma *fidlc-vi[simp]: is-fidlc-vi vi ⇒ Var-Int (fidlc-vi vi) = vi*
 ⟨proof⟩

lemma *fidlc-vv[simp]: ¬ is-fidlc-vi vv ⇒ Var-Var (fidlc-vv vv) = vv*
 ⟨proof⟩

datatype *'v fdl-constraints = IDL-CS ('v × int)list ('v × 'v)list 'v fdl-constraint list list*

fun *fdl-flat-cs :: 'v fdl-constraints ⇒ 'v fdl-constraint set set where fdl-flat-cs (IDL-CS vis vws cs) = set (map set (map (singleton o Var-Int) vis @ map (singleton o Var-Var) vws @ cs))*

fun *fdl-cs-restructure :: 'v fdl-constraints ⇒ 'v fdl-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⟩

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)) = (

$\text{map-prod id (Cons v) (fidl-delete-vi (IDL-State (Mapping.map-entry v (remove1 i) bnds) (IDL-CS vis vvs cs))))}$
 $| \text{fidl-delete-vi (IDL-State bnds (IDL-CS [] vvs cs))} = ($
 $\text{(IDL-State bnds (IDL-CS [] vvs cs)), [])}$

lemma *mapping-size-map-entry[simp]*: $\text{Mapping.size (Mapping.map-entry x f m)}$
 $= \text{Mapping.size m}$
 $\langle \text{proof} \rangle$

lemma *fidl-delete-vi: assumes* $\text{fidl-delete-vi } C = (C', xs) \text{ fidl-state } C \ V$
shows $\text{fidl-state } C' \ (V \cup \text{set } xs) \text{ fidl-state-sat } C' = \text{fidl-state-sat } C$
 $\langle \text{proof} \rangle$

lemma *fidl-delete-vi-size*: $\text{fidl-delete-vi } C = (C', vs) \implies \text{fidl-size } C' \leq \text{fidl-size } C$
 $\langle \text{proof} \rangle$

lemma *fidl-restructure*: $\text{fidl-restructure } s = \text{None} \implies \neg \text{Ex (fidl-state-sat } s)$
 $\text{fidl-restructure } s = \text{Some } s' \implies \text{fidl-state-sat } s' = \text{fidl-state-sat } s$
 $\text{fidl-restructure } s = \text{Some } s' \implies \text{fidl-state } s' = \text{fidl-state } s$
 $\text{fidl-restructure } s = \text{Some } s' \implies \text{fidl-size } s' = \text{fidl-size } s$
 $\langle \text{proof} \rangle$

lemma *all-entries-eq-all-lookups*:
 $(\forall (x, i) \in \text{Mapping.entries } m. P \ x \ i) = (\forall x \ i. \text{Mapping.lookup } m \ x = \text{Some } i \longrightarrow P \ x \ i)$
 $\langle \text{proof} \rangle$

fun *inst-vv where* $\text{inst-vv } v \ i \ [] = \text{Some } ([], [])$
 $| \text{inst-vv } v \ i \ ((x, y) \# xs) = (\text{if } x = v \ \text{then } (\text{if } y = v \ \text{then } \text{None}$
 $\text{else } \text{map-option (map-prod (Cons (y, i)) id) (inst-vv } v \ i \ xs))$
 $\text{else } \text{if } y = v \ \text{then } \text{map-option (map-prod (Cons (x, i)) id) (inst-vv } v \ i \ xs)$
 $\text{else } \text{map-option (map-prod id (Cons (x, y))) (inst-vv } v \ i \ xs))$

lemma *inst-vv: assumes* $\alpha \ v = i$
shows $\text{inst-vv } v \ i \ vvs = \text{None} \implies \neg \text{Ball (set (map Var-Var vvs)) (fidlc-sat } \alpha)$
 $\text{inst-vv } v \ i \ vvs = \text{Some } (vis, nvvs) \implies \text{Ball (set (map Var-Var vvs)) (fidlc-sat } \alpha)$
 $= (\text{Ball (set (map Var-Int vis)) (fidlc-sat } \alpha) \wedge \text{Ball (set (map Var-Var nvvs)) (fidlc-sat } \alpha))$
 $\text{inst-vv } v \ i \ vvs = \text{Some } (vis, nvvs) \implies \text{set (concat (map (fidlc-vars o Var-Int) vis))} \cup \text{set (concat (map (fidlc-vars o Var-Var) nvvs))}$
 $\subseteq \text{set (concat (map (fidlc-vars o Var-Var) vvs))} - \{v\}$
 $\langle \text{proof} \rangle$

fun *inst-fidlc* :: $'v \Rightarrow \text{int} \Rightarrow 'v \text{ fidl-constraint list} \Rightarrow 'v \text{ fidl-constraint list option}$
where
 $\text{inst-fidlc } v \ i \ [] = \text{Some } []$

| *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 *fdl-constraint list list* ⇒ 'v *fdl-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** $\alpha v = i$
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*⟩

lemma *inst-fidlc-list*: **assumes** $\alpha v = i$
shows *Ball* (*set* (*inst-fidlc-list v i ccs*)) ($\lambda cs. Bex$ (*set cs*) (*fidlc-sat* α))
= *Ball* (*set ccs*) ($\lambda cs. Bex$ (*set cs*) (*fidlc-sat* α))
set (*concat* (*concat* (*map* (*map fidlc-vars*) (*inst-fidlc-list v i ccs*)))) ⊆ *set* (*concat*
(*concat* (*map* (*map fidlc-vars*) *ccs*))) - {*v*}
⟨*proof*⟩

fun *instantiate-var* :: 'v ⇒ int ⇒ 'v *fdl-solver-state* ⇒ 'v *fdl-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*)
⟨*proof*⟩

lemma *instantiate-var*: **assumes** $\alpha v = i$

```

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  $\implies \neg$  fidl-state-sat (IDL-State bnds css)  $\alpha$ 
      so = Some s  $\implies$  fidl-state s V  $\wedge$  fidl-state-sat (IDL-State bnds css)  $\alpha$  = fidl-state-sat
      s  $\alpha$ 
       $\wedge$  fidl-vars s = fidl-vars (IDL-State bnds css) - {v}
       $\wedge$  fidl-size s < fidl-size (IDL-State bnds css)
<proof>

```

```

lemma instantiate-var-size: assumes instantiate-var v i s = Some s'
shows fidl-size s'  $\leq$  fidl-size s
<proof>

```

```

fun fidl-cs-empty :: 'v fidl-solver-state  $\Rightarrow$  bool where
  fidl-cs-empty (IDL-State bnds (IDL-CS [] [])) = True
| fidl-cs-empty - = False

```

```

lemma fidl-cs-empty: assumes fidl-state s {}
and fidl-cs-empty s
shows Ex (fidl-state-sat s)
<proof>

```

```

lemma fidl-vars: assumes  $\bigwedge v. v \in$  fidl-vars s  $\implies \alpha$  v =  $\beta$  v
and fidl-state s V
shows fidl-state-sat s  $\alpha$  = fidl-state-sat s  $\beta$ 
<proof>

```

```

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  $\implies$  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>

```

```

lemma clean-bnds-size: assumes clean-bnds s xs = Inl s'
shows fidl-size s'  $\leq$  fidl-size s

```

<proof>

definition *fidl-init* :: *bool* \Rightarrow $((v \times s) \times int)$ *list* \Rightarrow $((v \times s) \times v \times s)$ *list list*
 $\Rightarrow (v \times s)$ *fidl-solver-state* + *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 ($\lambda (vs,b).$ (*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*, if *Mapping.lookup sToV* (*snd*
vs) = *Some vs* then [*b*] 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*
<proof>

term *fidl-init*

lemma *fidl-init*: **assumes** *fidl-init sym-break* *bnds* *diffs* = *res*

and *fidl-input* $((bnds, diffs) :: (v,s)$ *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>

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* \Rightarrow (case *fidl-restructure s2* of

None \Rightarrow *Inr False*

 | *Some s3* \Rightarrow *Inl s3*)))

lemma *deduction-step*: **assumes** *fidl*: *fidl-state s* {}

and *res*: *deduction-step s* = *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>

lemma *deduction-step-size*: **assumes** *deduction-step s* = *Inl s'*

shows *fidl-size s'* \leq *fidl-size s*

<proof>

```

fun deduction-steps-main :: nat  $\Rightarrow$  'v fidl-solver-state  $\Rightarrow$  'v fidl-solver-state + bool
where
  deduction-steps-main n s = (case deduction-step s of
    Inr b  $\Rightarrow$  Inr b
  | Inl s'  $\Rightarrow$  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  $\Longrightarrow$  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>

lemma deduction-steps-size: assumes deduction-steps s = Inl s'
shows fidl-size s'  $\leq$  fidl-size s
  <proof>

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  $\Rightarrow$  ('v  $\times$  int option) + bool where
  fidl-find-var (IDL-State bnds (IDL-CS vis vws cs)) = (
    case vis of
      vi # vis2  $\Rightarrow$  Inl (map-prod id Some vi)
    | -  $\Rightarrow$  (case vws of vw # vws2  $\Rightarrow$  Inl (fst vw, None)
    | -  $\Rightarrow$  (case cs of []  $\Rightarrow$  Inr True
    | c # cs2  $\Rightarrow$  (case c of []  $\Rightarrow$  Inr False | a # as  $\Rightarrow$  Inl (fidl-var-int a))))))

fun reorder :: int option  $\Rightarrow$  int list  $\Rightarrow$  int list where
  reorder None xs = xs
  | reorder (Some i) xs = (case span ((<) i) xs of
    (bef, j # aft)  $\Rightarrow$  if i = j then bef @ aft @ [j] else xs
  | -  $\Rightarrow$  xs)

lemma set-reorder[simp]: set (reorder io xs) = set xs
  <proof>

definition finite-mapping m = finite (Mapping.keys m)

lemma finite-mapping-code[code]: finite-mapping (Mapping m) = True
  <proof>

function fidl-main-solver :: 'v fidl-solver-state  $\Rightarrow$  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))))))
⟨proof⟩

```

termination

⟨proof⟩

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)*
⟨proof⟩

definition *inner-solver sym-break* = (λ (bnds, diffs). case *fidl-init sym-break bnds*
diffs
of *Inl s ⇒ fidl-main-solver s* | *Inr b ⇒ b*)

lemma *inner-solver: finite-idl-solver (inner-solver sym-break)*
⟨proof⟩

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

⟨proof⟩

definition *default-fidl-solver* = *parametric-fidl-solver True True*

lemma *default-fidl-solver: finite-idl-solver default-fidl-solver*
⟨proof⟩

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* :: $- \Rightarrow - \Rightarrow ((f \times 's \text{ list}) \times 's) \text{ list} \Rightarrow (f, 'v, 's) \text{ pats-problem-list} \Rightarrow \text{bool}$ **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: snd  $\cup$  (vars  $\text{fst}$   $\text{set}$  (concat (concat P)))  $\subseteq$  set (sorts-of-ssig-list Cs)
and ren: renaming-funs rn rv
shows decide-pat-complete rn rv Cs P  $\longleftrightarrow$  pats-complete (map-of Cs) (pat-list  $\text{set}$  P)
 $\langle$ proof $\rangle$ 

```

export-code *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 *Pattern-Completeness*

```

imports
  Pattern-Completeness-List
  Pattern-Completeness-Improved-Algorithm
  Show.Shows-Literal
  Certification-Monads.Check-Monad
  Sorted-Terms.Basic-Terms

```

begin

lemmas [*iff del*] = *domIff*

A pattern completeness decision procedure for a set of lhss

definition *matches* :: $(f, 'v) \text{ term} \Rightarrow (f, 'w) \text{ term} \Rightarrow \text{bool}$ (**infix** \langle *matches* \rangle 50)

where

$l \text{ matches } t = (\exists \sigma. t = l \cdot \sigma)$

lemma *matches-subst*: $l \text{ matches } t \implies l \text{ matches } t \cdot \sigma$

<proof>

definition *pat-complete-lhss* :: $(f, 's) \text{ssig} \Rightarrow (f, 's) \text{ssig} \Rightarrow (f, 'v) \text{term set} \Rightarrow \text{bool}$

where

$\text{pat-complete-lhss } C D L = (\forall t \in \text{dom } \mathcal{T}_B(C, D). \exists l \in L. l \text{ matches } t)$

lemma *pat-complete-lhssD*:

assumes *comp*: $\text{pat-complete-lhss } C D L$ **and** $t: t \in \text{dom } \mathcal{T}_B(C, D, \emptyset)$

shows $\exists l \in L. l \text{ matches } t$

<proof>

definition *pats-of-lhss* :: $((f \times 's \text{ list}) \times 's) \text{list} \Rightarrow (f, 'v) \text{term list} \Rightarrow (f, 'v, 's) \text{pat-problem-list list}$ **where**

$\text{pats-of-lhss } D \text{lhss} = (\text{let } \text{pats} = [\text{Fun } f \text{ (map Var (zip [0..<length ss] ss))}. ((f, ss), s) \leftarrow D]$
 $\text{in } [[[(\text{pat}, \text{lhs})]. \text{lhs} \leftarrow \text{lhss}]. \text{pat} \leftarrow \text{pats}])$

definition *check-signatures* :: $((f \times 's \text{ list}) \times 's) \text{list} \Rightarrow ((f \times 's \text{ list}) \times 's) \text{list} \Rightarrow \text{showsl check}$ **where**

$\text{check-signatures } C D = \text{do } \{$
 $\text{check } (\text{distinct } (\text{map fst } C)) (\text{showsl-lit } (\text{STR } \text{"constructor information contains duplicate"}));$
 $\text{check } (\text{distinct } (\text{map fst } D)) (\text{showsl-lit } (\text{STR } \text{"defined symbol information contains duplicate"}));$
 $\text{let } S = \text{sorts-of-ssig-list } C;$
 $\text{check-allm } (\lambda ((f, ss), -). \text{check-allm } (\lambda s. \text{check } (s \in \text{set } S)$
 $(\text{showsl-lit } (\text{STR } \text{"a defined symbol has argument sort that is not known in constructors"}))) ss) D;$
 $(\text{case } (\text{decide-nonempty-sorts } S C) \text{ of None} \Rightarrow \text{return } () \mid \text{Some } s \Rightarrow \text{error}$
 $(\text{showsl-lit } (\text{STR } \text{"some sort is empty"})))$
 $\}$

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**

$\text{decide-pat-complete-linear-lhss } C D \text{lhss} = \text{do } \{$
 $\text{check-signatures } C D;$
 $\text{return } (\text{decide-pat-complete-lin } C (\text{pats-of-lhss } D \text{lhss}))$
 $\}$

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**

$\text{decide-pat-complete-lhss-fscd } C D \text{lhss} = \text{do } \{$

```

    check-signatures C D;
    return (decide-pat-complete-fscd C (pats-of-lhss D lhss))
  }

```

definition *decide-pat-complete-lhss* ::

- \Rightarrow - \Rightarrow $((f \times 's \text{ list}) \times 's) \text{ list} \Rightarrow ((f \times 's \text{ list}) \times 's) \text{ list} \Rightarrow (f, 'v) \text{ term list} \Rightarrow$
showsl + *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 $\text{snd} \ ' \cup (\text{vars} \ ' \text{fst} \ ' \text{set} (\text{concat} (\text{concat} (\text{pats-of-lhss } D \text{ lhss})))) \subseteq S$
<proof>

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>

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>

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 = *return* b

shows $b = \text{pat-complete-lhss} (\text{map-of } C) (\text{map-of } D) (\text{set } \text{lhss})$

<proof>

theorem *decide-pat-complete-linear-lhss*:

fixes C D :: $((f \times 's \text{ list}) \times 's) \text{ list}$ **and** lhss :: $(f, 'v) \text{ term list}$

assumes *decide-pat-complete-linear-lhss* C D lhss = *return* b

and *linear*: *Ball* (*set* lhss) *linear-term*

shows $b = \text{pat-complete-lhss} (\text{map-of } C) (\text{map-of } D) (\text{set } \text{lhss})$

<proof>

theorem *decide-pat-complete-lhss*:

fixes C D :: $((f \times 's \text{ list}) \times 's) \text{ list}$ **and** lhss :: $(f, 'v) \text{ term list}$

assumes *decide-pat-complete-lhss* rn rv C D lhss = *return* b

and *ren*: *renaming-funs* rn rv

shows $b = \text{pat-complete-lhss } (\text{map-of } C) (\text{map-of } D) (\text{set lhss})$
 ⟨proof⟩

Definition of strong quasi-reducibility and a corresponding decision procedure

definition *strong-quasi-reducible* :: $(f, 's)\text{ssig} \Rightarrow (f, 's)\text{ssig} \Rightarrow (f, 'v)\text{term set} \Rightarrow \text{bool}$ **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 $f ts = \text{Fun } f ts \# ts$

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 $rn rv C D lhss = \text{do } \{$
 $\text{check-signatures } C D;$
 $\text{let pats} = \text{map } (\lambda ((f, ss), s). \text{term-and-args } f (\text{map Var } (\text{zip } [0..<\text{length } ss] ss)))$
 $D;$
 $\text{let } P = \text{map } (\text{List.maps } (\lambda \text{pat}. \text{map } (\lambda \text{lhs}. [(pat, lhs)]) lhss)) pats;$
 $\text{return } (\text{decide-pat-complete } rn rv C P)$
 $\}$

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* $rn rv C D lhss = \text{return } b$

and $ren: \text{renaming-funs } rn rv$

shows $b = \text{strong-quasi-reducible } (\text{map-of } C) (\text{map-of } D) (\text{set lhss})$

⟨proof⟩

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

definition *quasi-reducible* :: $(f, 's)\text{ssig} \Rightarrow (f, 's)\text{ssig} \Rightarrow (f, 'v)\text{term set} \Rightarrow \text{bool}$ **where**

quasi-reducible $C D L = (\forall t \in \text{dom } \mathcal{T}_B(C, D). \exists tp \leq t. \exists l \in L. l \text{ matches } tp)$

lemma *pat-complete-imp-strong-quasi-reducible*:

pat-complete-lhss $C D L \Longrightarrow \text{strong-quasi-reducible } C D L$

⟨proof⟩

lemma *arg-imp-subt*: $s \in \text{set } (\text{args } t) \Longrightarrow t \supseteq s$

⟨proof⟩

lemma *strong-quasi-reducible-imp-quasi-reducible*:

strong-quasi-reducible $C D L \Longrightarrow \text{quasi-reducible } C D L$

⟨proof⟩

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** $L :: ('f, 'v)\text{term set}$
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 $\text{pat-complete-lhss } C \ D \ L \longleftrightarrow \text{quasi-reducible } C \ D \ L$
<proof>

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 $C \ D \ \text{lhss} =$ (
case decide-pat-complete-lhss-fscd $C \ D \ \text{lhss}$ of *Inl* $\text{err} \Rightarrow \text{Code.abort } (\text{err } (\text{STR}$
 $\text{''}'')) (\lambda \text{ . True})$
 $| \text{Inr } \text{res} \Rightarrow \text{res})$

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 $\text{rn } \text{rv } C \ D \ \text{lhss} =$ (
case decide-strong-quasi-reducible $\text{rn } \text{rv } C \ D \ \text{lhss}$ of *Inl* $\text{err} \Rightarrow \text{Code.abort } (\text{err } (\text{STR}$
 $\text{''}'')) (\lambda \text{ . True})$
 $| \text{Inr } \text{res} \Rightarrow \text{res})$

Examples

definition *nat-bool* = [
 $(\text{''zero''}, [], \text{''nat''}),$
 $(\text{''succ''}, [\text{''nat''}], \text{''nat''}),$
 $(\text{''true''}, [], \text{''bool''}),$
 $(\text{''false''}, [], \text{''bool''})$
 $]$

definition *rn-string* **where** *rn-string* $x = "x" @ \text{show } (x :: \text{nat})$

definition *rv-string* **where** *rv-string* $x = "y" @ x$

lemma *renaming-string*: *renaming-funs rn-string rv-string*
{*proof*}

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

<proof>

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

<proof>

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

<proof>

lemma pat-complete-lhss (map-of nat-bool) (map-of even-nat) (set even-lhss)

<proof>

lemma \neg pat-complete-lhss (map-of int-bool) (map-of even-int) (set even-lhss-int)

<proof>

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)

<proof>

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)

<proof>

lemma \neg pat-complete-lhss (map-of nat-bool) (map-of [((("f",["nat","nat","nat"]),"bool"))])

(set non-lin-lhss)

<proof>

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)
 ⟨proof⟩

lemma \neg *pat-complete-lhss* (map-of nat-bool) (map-of [((('f',["nat","nat","nat"]),"bool"))
 (set non-lin-lhss)
 ⟨proof⟩

lemma *pat-complete-lhss* (map-of nat-bool) (map-of [((('f',["bool","bool","bool"]),"bool"))
 (set non-lin-lhss)
 ⟨proof⟩

definition *testproblem* (c :: nat) n = (let s = String.implode; s = id;

 c1 = even c;
 c2 = even (c div 2);
 c3 = even (c div 4);
 c4 = even (c div 8);
 revo = (if c4 then id else rev);
 nn = [0 ..< n];
 rnn = (if c4 then id nn else rev nn);
 b = s "b"; t = s "tt"; f = s "ff"; g = s "g";
 gg = (λ ts. Fun g (revo ts));
 ff = Fun f [];
 tt = Fun t [];
 C = [(t, [] :: string list), b], ((f, []), b);
 D = [(g, replicate (2 * n) b), b];
 x = (λ i :: nat. Var (s "x" @ show i));
 y = (λ i :: nat. Var (s "y" @ show i));
 lhsF = gg (if c1 then List.maps (λ i. [ff, y i]) rnn else (replicate n ff @ map
 y rnn));
 lhsT = (λ b j. gg (if c1 then List.maps (λ i. if i = j then [tt, b] else [x i, y i])
 rnn else
 (map (λ i. if i = j then tt else x i) rnn @ map (λ i. if i = j then b else
 y i) rnn));
 lhssT = (if c2 then List.maps (λ i. [lhsT tt i, lhsT ff i]) nn else List.maps (λ
 b. map (lhsT b) nn) [tt,ff]);
 lhss = (if c3 then [lhsF] @ lhssT else lhssT @ [lhsF])
 in (C, D, lhss))

definition *test-problem* c n perms = (if c < 16 then testproblem c n
 else let (C, D, lhss) = testproblem 0 n;
 (permRow,permCol) = perms ! (c - 16);
 permRows = map (λ i. lhss ! i) permRow;

$pCol = (\lambda t. \text{case } t \text{ of } Fun \ g \ ts \Rightarrow Fun \ g \ (\text{map } (\lambda i. \ ts \ ! \ i) \ permCol))$
in $(C, D, \text{map } pCol \ permRows)$

definition *test-problem-integer where*

test-problem-integer $c \ n \ perms = \text{test-problem } (\text{nat-of-integer } c) \ (\text{nat-of-integer } n) \ (\text{map } (\text{map-prod } (\text{map } \text{nat-of-integer}) \ (\text{map } \text{nat-of-integer})) \ perms)$

fun *term-to-haskell where*

term-to-haskell $(Var \ x) = String.implode \ x$
| *term-to-haskell* $(Fun \ f \ ts) = (\text{if } f = "tt" \ \text{then } STR \ "TT" \ \text{else if } f = "ff" \ \text{then } STR \ "FF" \ \text{else } String.implode \ f)$
+ *foldr* $(\lambda t \ r. \ STR \ " " + \text{term-to-haskell } t + r) \ ts \ (STR \ ""')$

definition *createHaskellInput :: integer => integer => (integer list * integer list) list => String.literal where*

createHaskellInput $c \ n \ perms = (\text{case } \text{test-problem-integer } c \ n \ perms \ \text{of}$
 $(-, -, lhss) \Rightarrow STR \ "module \ Test(g) \ \text{where } \boxed{\leftrightarrow} \boxed{\leftrightarrow} \ \text{data } B = TT \ | \ FF \boxed{\leftrightarrow} \boxed{\leftrightarrow}"$
+
foldr $(\lambda l \ s. \ (\text{term-to-haskell } l + STR \ " = TT \boxed{\leftrightarrow}" + s)) \ lhss \ (STR \ ""')$

definition *pat-complete-alg-test-fscd :: integer => integer => (integer list * integer list) list => bool where*

pat-complete-alg-test-fscd $c \ n \ perms = (\text{case } \text{test-problem-integer } c \ n \ perms \ \text{of}$
 $(C, D, lhss) \Rightarrow \text{pat-complete-alg-fscd } C \ D \ lhss)$

definition *show-pat-complete-test :: integer => integer => (integer list * integer list) list => String.literal where*

show-pat-complete-test $c \ n \ perms = (\text{case } \text{test-problem-integer } c \ n \ perms \ \text{of } (-, -, lhss)$
 $\Rightarrow \text{showsl-lines } (STR \ "empty") \ lhss \ (STR \ ""')$

definition *create-agcp-input :: (String.literal => 't) => integer => integer => (integer list * integer list) list =>*

*'t list list * 't list list where*

create-agcp-input $term \ C \ N \ perms = (\text{let}$
 $n = \text{nat-of-integer } N;$
 $c = \text{nat-of-integer } C;$
 $lhss = (\text{snd } o \ \text{snd}) \ (\text{test-problem-integer } C \ N \ perms);$
 $tt = (\lambda t. \ \text{case } t \ \text{of } (Var \ x) \Rightarrow \text{term } (String.implode \ ("?" @ x @ ":B"))$
| $Fun \ f \ [] \Rightarrow \text{term } (String.implode \ f));$
 $pslist = \text{map } (\lambda i. \ tt \ (Var \ ("x" @ \text{show } i))) \ [0..< 2 * n];$

$patlist = \text{map } (\lambda t. \ \text{case } t \ \text{of } Fun \ - \ ps \Rightarrow \text{map } tt \ ps) \ lhss$
in $([pslist], patlist)$

tree automata encoding

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

context

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

we further fix the parameters to generate the example TRSs

context

fixes $c n :: integer$
and $perms :: (integer list \times integer list) list$
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 $tRule = (\lambda q. cRule tt [] q)$
definition $fRule = (\lambda q. cRule ff [] q)$

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**

$argToState (Var -) = qb$
 $| argToState (Fun s []) = (if s = "tt" then qt else if s = "ff" then qf$
 $else Code.abort (STR "unknown") (\lambda -. qf))$

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 $automataRight = (case test-problem-integer c n perms of$
 $(-, -, lhss) \Rightarrow cAut signature rightStates finStates (map termToRule lhss @ stdRules))$

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` \rightarrow
`(Haskell) infixr 5 ++`

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

definition `showAutomata where showAutomata n c perms = (case encodeAutomata id (\ n a. n)`
`(\ f qs q. paren f (f +++ STR "(") (STR ")") (STR ",") qs +++ STR " ->`
`" +++ q)`
`(\ 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`
`""')) (\ -. 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`
`""')) (\ -. True)`

| *Inr res* \Rightarrow *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* \Rightarrow *integer* \Rightarrow (*integer list* * *integer list*)*list* \Rightarrow *bool* **where**

pat-complete-alg-test-linear c n perms = (*case test-problem-integer c n perms of*
(C,D,lhss) \Rightarrow *pat-complete-alg-linear C D lhss*)

definition *pat-complete-alg-test-new* :: *integer* \Rightarrow *integer* \Rightarrow (*integer list* * *integer list*)*list* \Rightarrow *bool* **where**

pat-complete-alg-test-new c n perms = (*case test-problem-integer c n perms of*
(C,D,lhss) \Rightarrow *pat-complete-alg-new C D lhss*)

definition *test-problem-nl1* :: *integer* \Rightarrow - **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* \Rightarrow - **where**

test-problem-nl2 n = (*case test-problem-nl1 n of*
(C, D, lhss) \Rightarrow ((*d''*, []), *s0''*) # *C, D, lhss*)

definition *test-problem-nl3* :: *integer* \Rightarrow - **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 *" "* +
shows 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 ⟨proof⟩

value(code) showsl (test-problem-nl2 2) (STR ""')

lemma ¬ pat-complete-alg-test-nl-new 2 2 ⟨proof⟩

value(code) showsl (test-problem-nl3 2) (STR ""')

lemma pat-complete-alg-test-nl-new 3 2 ⟨proof⟩

value(code) showsl (test-problem-nl4 2) (STR ""')

lemma ¬ pat-complete-alg-test-nl-new 4 2 ⟨proof⟩

value(code) showsl (test-problem-nl5 2) (STR ""')

lemma pat-complete-alg-test-nl-new 5 2 ⟨proof⟩

value(code) showsl (test-problem-nl6 2) (STR ""')

lemma ¬ pat-complete-alg-test-nl-new 6 2 ⟨proof⟩

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

<proof>

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.