Verifying a Decision Procedure for Pattern Completeness*

René Thiemann

University of Innsbruck, Austria

Akihisa Yamada

National Institute of Advanced Industrial Science and Technology, Japan

March 17, 2025

Abstract

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

Contents

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

1 Introduction

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

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

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

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

theory *Singleton-List*
imports *Main*
begin

definition *singleton* $x = [x]$

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

lemma *is-singleton-list*: *is-singleton-list xs* ⟷ *set* (*singleton* (*hd xs*)) = *set xs*
⟨*proof*⟩

lemma *is-singleton-list2*: *is-singleton-list xs* ⟷ (∃ 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* × *'s*) × *int*) *list* × ((*'v* × *'s*) × *'v* × *'s*) *list list*

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

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

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

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

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

lemma *dummy-fidl-solver*: *finite-idl-solver dummy-fidl-solver*
⟨*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 Computing Nonempty and Infinite sorts

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

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

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

⟨*proof*⟩

lemma *finite-listset*:
assumes $\forall A \in set\ As.\ finite\ A$
shows $finite\ (listset\ As)$
⟨*proof*⟩

lemma *listset-conv-nth*:
$xs \in listset\ As = (length\ xs = length\ As \wedge (\forall i < length\ As.\ xs\ !\ i \in As\ !\ i))$
⟨*proof*⟩

lemma *card-listset*: **assumes** $\bigwedge A.\ A \in set\ As \Longrightarrow finite\ A$
shows $card\ (listset\ As) = prod\text{-}list\ (map\ card\ As)$
⟨*proof*⟩

4.1 Deciding the nonemptyness of all sorts under consideration

function *compute-nonempty-main* :: $'\tau\ set \Rightarrow (('f \times '\tau\ list) \times '\tau)\ list \Rightarrow '\tau\ set$
where
$compute\text{-}nonempty\text{-}main\ ne\ ls = (let\ rem\text{-}ls = filter\ (\lambda\ f.\ snd\ f \notin ne)\ ls\ in$
$case\ partition\ (\lambda\ ((\text{-},args),\text{-}).\ set\ args \subseteq ne)\ rem\text{-}ls\ of$
$(new,\ rem) \Rightarrow if\ new = []\ then\ ne\ else\ compute\text{-}nonempty\text{-}main\ (ne \cup set\ (map\ snd\ new))\ rem)$
⟨*proof*⟩

termination
⟨*proof*⟩

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

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

lemma *compute-nonempty-sorts*:
assumes $distinct\ (map\ fst\ Cs)$
shows $compute\text{-}nonempty\text{-}sorts\ Cs = \{\tau.\ \neg\ empty\text{-}sort\ (map\text{-}of\ Cs)\ \tau\}$ (**is** - = *?NE*)
⟨*proof*⟩

definition *decide-nonempty-sorts* :: $'t\ list \Rightarrow (('f \times 't\ list) \times 't)list \Rightarrow 't\ option$
where
$decide\text{-}nonempty\text{-}sorts\ \tau s\ Cs = (let\ ne = compute\text{-}nonempty\text{-}sorts\ Cs\ in$
$find\ (\lambda\ \tau.\ \tau \notin ne)\ \tau s)$

lemma *decide-nonempty-sorts*:
assumes $distinct\ (map\ fst\ Cs)$
shows $decide\text{-}nonempty\text{-}sorts\ \tau s\ Cs = None \Longrightarrow \forall\ \tau \in set\ \tau s.\ \neg\ empty\text{-}sort\ (map\text{-}of\ Cs)\ \tau$

decide-nonempty-sorts τs *Cs* = *Some* $\tau \Longrightarrow \tau \in$ *set* τs $\wedge$ *empty-sort* (*map-of Cs*) τ
⟨*proof*⟩

4.2 Deciding infiniteness of a sort and computing cardinalities

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

context
includes *finfun-syntax*
begin

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

lemma *finfun-update-all*[*simp*]: *finfun-update-all xs g f* $ *x* = (*if x* ∈ *set xs then g x else f* $ *x*)
⟨*proof*⟩

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

function *compute-inf-card-main* :: *'τ set* ⇒ (*'τ* ⇒f *nat*) ⇒ (*'τ* × (*'f* × *'τ list*)*list*) *list* ⇒ *'τ set* × (*'τ* ⇒ *nat*) **where**
compute-inf-card-main m-inf cards ls = (
let (*fin*, *ls'*) =
partition (λ (τ,*fs*). ∀ τ*s* ∈ *set* (*map snd fs*). ∀ τ ∈ *set* τ*s*. τ ∉ *m-inf*) *ls*
in if fin = [] *then* (*m-inf*, λ τ. *cards* $ τ) *else*
let new = *map fst fin*;
cards' = *finfun-update-all new* (λ τ. *compute-card-of-sort* τ (*the* (*map-of ls* τ)) *cards*) *cards in*
compute-inf-card-main (*m-inf* − *set new*) *cards' ls'*)
⟨*proof*⟩

termination
⟨*proof*⟩

lemma *compute-inf-card-main*: **fixes** *C* :: (*'f*,*'t*)*ssig*
assumes *C-Cs*: *C* = *map-of Cs'*

and *Cs'*: *set Cs'* = *set* (*concat* (*map* ((λ (τ, *fs*). *map* (λ *f*. (*f*,τ)) *fs*)) *Cs*))
and *arg-types-nonempty*: ∀ *f* τ*s* τ τ'. *f* : τ*s* → τ *in C* ⟶ τ' ∈ *set* τ*s* ⟶ ¬ *empty-sort C* τ'
and *dist*: *distinct* (*map fst Cs*) *distinct* (*map fst Cs'*)
and *inhabitet*: ∀ τ *fs*. (τ,*fs*) ∈ *set Cs* ⟶ *set fs* ≠ {}
and ∀ τ. τ ∉ *m-inf* ⟶ *bdd-above* (*size* ‘ {*t*. *t* : τ *in* $\mathcal{T}(C)$})
and *set ls* ⊆ *set Cs*
and *fst* ‘ (*set Cs* − *set ls*) ∩ *m-inf* = {}
and *m-inf* ⊆ *fst* ‘ *set ls*
and ∀ τ. τ ∉ *m-inf* ⟶ *cards* $ τ = *card-of-sort C* τ ∧ *finite-sort C* τ
and ∀ τ. τ ∈ *m-inf* ⟶ *cards* $ τ = *0*
shows *compute-inf-card-main m-inf cards ls* = ({τ. ¬ *bdd-above* (*size* ‘ {*t*. *t* : τ *in* $\mathcal{T}(C)$})},
λ τ. *card-of-sort C* τ)
⟨*proof*⟩

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

lemma *finite-imp-size-bdd-above*: **assumes** *finite T*
shows *bdd-above* (*size* ‘ *T*)
⟨*proof*⟩

lemma *finite-sig-imp-finite-terms-of-bounded-size*: **assumes** *finite* (*dom F*) **and** *finite* (*dom V*)
shows *finite* {*t*. ∃ τ. *size t* ≤ *n* ∧ *t* : τ *in* $\mathcal{T}(F,V)$} (**is** *finite* (*?terms n*))
⟨*proof*⟩

lemma *finite-sig-bdd-above-imp-finite*: **assumes** *finite* (*dom F*) **and** *finite* (*dom V*)
and *bdd-above* (*size* ‘ {*t*. *t* : τ *in* $\mathcal{T}(F,V)$})
shows *finite* {*t*. *t* : τ *in* $\mathcal{T}(F,V)$}
⟨*proof*⟩

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

shows *bdd-above* (*size* ‘ {*t*. *t* : τ *in* $\mathcal{T}(F,V)$}) = *finite* {*t*. *t* : τ *in* $\mathcal{T}(F,V)$}
⟨*proof*⟩

lemma *compute-inf-card-sorts*:
fixes *C* :: (*'f*,*'t*)*ssig*
assumes *C-Cs*: *C* = *map-of Cs*
and *arg-types-nonempty*: ∀ *f* τ*s* τ τ'. *f* : τ*s* → τ *in C* ⟶ τ' ∈ *set* τ*s* ⟶ ¬

empty-sort C τ'
  and *dist*: *distinct* (*map fst Cs*)
  and *result*: *compute-inf-card-sorts Cs* = (*unb*, *cards*)
shows *unb* = $\{\tau.\ \neg$ *bdd-above* (*size* ` $\{t.\ t : \tau$ *in* $\mathcal{T}(C)\})\}$ (**is** - = *?unb*)
  cards = *card-of-sort C* (**is** - = *?cards*)
  unb = $\{\tau.\ \neg$*finite-sort C* $\tau\}$ (**is** - = *?inf*)
⟨*proof*⟩
end

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

lemma *compute-inf-sorts*:
  assumes *arg-types-nonempty*: ∀ *f* $\tau s\ \tau\ \tau'$. *f* : $\tau s \to \tau$ *in map-of Cs* ⟶ $\tau' \in$ *set* τs ⟶ ¬ *empty-sort* (*map-of Cs*) τ'
  and *dist*: *distinct* (*map fst Cs*)
shows
  compute-inf-sorts Cs = $\{\tau.\ \neg$ *bdd-above* (*size* ` $\{t.\ t : \tau$ *in* $\mathcal{T}$(*map-of Cs*)$\})\}$
  compute-inf-sorts Cs = $\{\tau.\ \neg$ *finite-sort* (*map-of Cs*) $\tau\}$
  ⟨*proof*⟩

end

5 Pattern Completeness

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

6 A Set-Based Inference System to Decide Pattern Completeness

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

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

theory *Pattern-Completeness-Set*
  imports
    First-Order-Terms.Term-More
    Complete-Non-Orders.Complete-Relations
    Sorted-Terms.Sorted-Contexts
    Compute-Nonempty-Infinite-Sorts

begin

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

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

lemma *bex-insert-cong*: $f\ y = f\ z \Longrightarrow Bex\ (insert\ y\ A)\ f = Bex\ (insert\ z\ A)\ f$
⟨*proof*⟩

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

lemma *list-eq-nth-eq*: $xs = ys \longleftrightarrow length\ xs = length\ ys \wedge (\forall\ i < length\ ys.\ xs\ !\ i = ys\ !\ i)$
⟨*proof*⟩

lemma *subt-size*: $p \in poss\ t \Longrightarrow size\ (t\ |\text{-}\ p) \leq size\ t$
⟨*proof*⟩

lemma *removeAll-remdups*: *removeAll x* (*remdups ys*) = *remdups* (*removeAll x ys*)
⟨*proof*⟩

lemma *removeAll-eq-Nil-iff*: $removeAll\ x\ ys = [] \longleftrightarrow (\forall y \in set\ ys.\ y = x)$
⟨*proof*⟩

lemma *concat-removeAll-Nil*: *concat* (*removeAll* [] *xss*) = *concat xss*
⟨*proof*⟩

lemma *removeAll-eq-imp-concat-eq*:
assumes *removeAll* [] *xss* = *removeAll* [] *xss'*
shows *concat xss* = *concat xss'*
⟨*proof*⟩

lemma *map-remdups-commute*:
assumes *inj-on f* (*set xs*)
shows *map f* (*remdups xs*) = *remdups* (*map f xs*)
⟨*proof*⟩

lemma *Uniq-False*: $\exists_{\leq 1}\ a.\ False$ ⟨*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** $a = the\text{-}elem\ A$

⟨*proof*⟩

lemma *bij-betw-imp-Uniq-iff*:
assumes *bij-betw f A B* **shows** *UNIQ A* ⟷ *UNIQ B*
⟨*proof*⟩

lemma *image-Uniq*: *UNIQ A* ⟹ *UNIQ* (*f* ‘ *A*)
⟨*proof*⟩

lemma *successively-eq-iff-Uniq*: *successively* (=) *xs* ⟷ *UNIQ* (*set xs*) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

6.1 Defining Pattern Completeness

We first consider matching problems, which are set of matching atoms. Each matching atom is a pair of terms: matchee and pattern. Matchee and pattern may have different type of variables: Matchees use natural numbers (annotated with sorts) as variables, so that it is easy to generate new variables, whereas patterns allow arbitrary variables of type *'v* without any further information. Then pattern problems are sets of matching problems, and we also have sets of pattern problems.

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

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

type-synonym (*'f*,*'v*,*'s*)*match-atom* = (*'f*,*nat* × *'s*)*term* × (*'f*,*'v*)*term*
type-synonym (*'f*,*'v*,*'s*)*match-problem-set* = (*'f*,*'v*,*'s*) *match-atom set*
type-synonym (*'f*,*'v*,*'s*)*pat-problem-set* = (*'f*,*'v*,*'s*)*match-problem-set set*
type-synonym (*'f*,*'v*,*'s*)*pats-problem-set* = (*'f*,*'v*,*'s*)*pat-problem-set set*

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

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

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

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

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

A definition of pattern completeness for pattern problems.

definition *match-complete-wrt* :: $('f,nat \times 's,'w)gsubst \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}set$ $\Rightarrow$ *bool* **where**
match-complete-wrt σ *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-match mp* $\Longrightarrow \sigma\ x = \sigma'\ x$
and *mp*: *mp* = *mp'*
shows *match-complete-wrt* σ *mp* = *match-complete-wrt* σ' *mp'*
⟨*proof*⟩

lemma *match-complete-wrt-imp-o*:
assumes *match-complete-wrt* σ *mp* **shows** *match-complete-wrt* $(\sigma \circ_s \tau)$ *mp*
⟨*proof*⟩

lemma *match-complete-wrt-o-imp*:
assumes *s*: $\sigma :_s \mathcal{V}$ |' *tvars-match mp* $\to \mathcal{T}(C,\emptyset)$ **and** *m*: *match-complete-wrt* $(\sigma \circ_s \tau)$ *mp*
shows *match-complete-wrt* σ *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 $\longleftrightarrow (\forall\, \sigma :_s \mathcal{V}$ |' *tvars-pat pp* $\to \mathcal{T}(C).\ \exists\ mp \in pp.$ *match-complete-wrt* σ *mp*)

lemma *pat-completeD*:
assumes *pp*: *pat-complete C pp*
and *s*: $\sigma :_s \mathcal{V}$ |' *tvars-pat pp* $\to \mathcal{T}(C,\emptyset)$
shows $\exists\ mp \in pp.$ *match-complete-wrt* σ *mp*
⟨*proof*⟩

lemma *pat-completeI*:
assumes *r*: $\forall\, \sigma :_s \mathcal{V}$ |' *tvars-pat pp* $\to \mathcal{T}(C,\emptyset{::}'v \rightharpoonup 's).\ \exists\ mp \in pp.$ *match-complete-wrt* σ *mp*
shows *pat-complete C pp*
⟨*proof*⟩

lemma *tvars-pat-empty*[*simp*]: *tvars-pat* {} = {}
⟨*proof*⟩

lemma *pat-complete-empty*[*simp*]: *pat-complete C* {} = *False*
⟨*proof*⟩

abbreviation *pats-complete* :: $('f,'s)$ *ssig* $\Rightarrow ('f,'v,'s)pats\text{-}problem\text{-}set \Rightarrow$ *bool* **where**
pats-complete C P $\equiv \forall\, pp \in P.$ *pat-complete C pp*

6.2 Definition of Algorithm – Inference Rules

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

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

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

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

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

lemma *conflicts-sym*: *rel-option* (λ *xs ys*. *set xs* = *set ys*) (*conflicts s t*) (*conflicts t s*) (**is** *rel-option* - (*?c s t*) -)
⟨*proof*⟩

lemma *conflicts*:
 shows *Conflict-Clash s t* ⟹
  ∃ *p*. *p* ∈ *poss s* ∧ *p* ∈ *poss t* ∧
  (*is-Fun* (*s* |-*p*) ∧ *is-Fun* (*t* |-*p*) ∧ *root* (*s* |-*p*) ≠ *root* (*t* |- *p*) ∨
  (∃ *x y*. *s* |-*p* = *Var x* ∧ *t* |-*p* = *Var y* ∧ *snd x* ≠ *snd y*))
  (**is** *?B1* ⟹ *?B2*)
  and *Conflict-Var s t x* ⟹
  ∃ *p* . *p* ∈ *poss s* ∧ *p* ∈ *poss t* ∧ *s* |-*p* ≠ *t* |-*p* ∧
  (*s* |-*p* = *Var x* ∨ *t* |-*p* = *Var x*)
  (**is** *?C1 x* ⟹ *?C2 x*)
  and *s* ≠ *t* ⟹ ∃ *x*. *Conflict-Clash s t* ∨ *Conflict-Var s t x*
  and *Conflict-Var s t x* ⟹ *x* ∈ *vars s* ∪ *vars t*
  and *conflicts s t* = *Some* [] ⟷ *s* = *t* (**is** *?A*)
⟨*proof*⟩

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

lemma *conflicts-refl*[*simp*]: *conflicts t t* = *Some* []
 ⟨*proof*⟩

locale *pattern-completeness-context* =
 fixes *S* :: *'s set* — set of sort-names

and C :: $('f,'s)ssig$ — sorted signature
and m :: nat — upper bound on arities of constructors
and Cl :: $'s \Rightarrow ('f \times 's\ list)list$ — a function to compute all constructors of given sort as list
and $inf\text{-}sort$ :: $'s \Rightarrow bool$ — a function to indicate whether a sort is infinite
and $cd\text{-}sort$ :: $'s \Rightarrow nat$ — a function to compute finite cardinality of a sort
and $improved$ :: $bool$ — if improved = False, then FSCD-version of algorithm is used; if improved = True, the better journal version (under development) is used.
begin

definition $tvars\text{-}disj\text{-}pp$ :: $nat\ set \Rightarrow ('f,'v,'s)pat\text{-}problem\text{-}set \Rightarrow bool$ **where**
$tvars\text{-}disj\text{-}pp\ V\ p = (\forall\ mp \in p.\ \forall\ (ti,pi) \in mp.\ fst\ `\ vars\ ti \cap V = \{\})$

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

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

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

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

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

The transformation rules of the paper.

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

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

inductive $mp\text{-}step$ :: $('f,'v,'s)match\text{-}problem\text{-}set \Rightarrow ('f,'v,'s)match\text{-}problem\text{-}set \Rightarrow bool$
(**infix** $\langle\to_s\rangle$ 50) **where**
$mp\text{-}decompose$: $length\ ts = length\ ls \Longrightarrow insert\ (Fun\ f\ ts,\ Fun\ f\ ls)\ mp \to_s set\ (zip\ ts\ ls) \cup mp$
| $mp\text{-}match$: $x \notin \bigcup\ (vars\ `\ snd\ `\ mp) \Longrightarrow insert\ (t,\ Var\ x)\ mp \to_s mp$
| $mp\text{-}identity$: $mp \to_s mp$
| $mp\text{-}decompose'$: $mp \cup mp' \to_s (\bigcup\ (t,\ l) \in mp.\ set\ (zip\ (args\ t)\ (map\ Var\ ys))) \cup mp'$
if $\bigwedge\ t\ l.\ (t,l) \in mp \Longrightarrow l = Var\ y \wedge root\ t = Some\ (f,n)$

$\bigwedge$ *t l.* (*t*,*l*) $\in$ *mp'* $\Longrightarrow$ *y* $\notin$ *vars l*
lvars-disj-mp ys (*mp* $\cup$ *mp'*) *length ys* = *n*
improved

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

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

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

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

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

inductive *P-step-set* :: ('*f*,'*v*,'*s*)*pats-problem-set* $\Rightarrow$ ('*f*,'*v*,'*s*)*pats-problem-set* $\Rightarrow$ *bool*
(**infix** ‹$\Rrightarrow_s$› *50*) **where**
P-fail: *insert* {} *P* $\Rrightarrow_s$ *bottom*
| *P-simp*: *pp* $\Rightarrow_s$ *pp'* $\Longrightarrow$ *insert pp P* $\Rrightarrow_s$ *insert pp' P*
| *P-remove-pp*: *pp-success pp* $\Longrightarrow$ *insert pp P* $\Rrightarrow_s$ *P*
| *P-instantiate*: *tvars-disj-pp* {*n* ..< *n*+*m*} *pp* $\Longrightarrow$ *x* $\in$ *tvars-pat pp* $\Longrightarrow$
insert pp P $\Rrightarrow_s$ {*subst-pat-problem-set* τ *pp* |. τ $\in$ τs *n x*} $\cup$ *P*

6.3 Soundness of the inference rules

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

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

lemma *wf-match-iff*: *wf-match mp* $\longleftrightarrow$ ($\forall$ (*x*,ι) $\in$ *tvars-match mp*. ι $\in$ *S*)

⟨*proof*⟩

lemma *tvars-match-subst*: *tvars-match* (*subst-match-problem-set* σ *mp*) = ($\bigcup$ (t,l) $\in$ *mp*. *vars* ($t \cdot \sigma$))
⟨*proof*⟩

lemma *wf-match-subst*:
assumes *s*: $\sigma :_s \mathcal{V}$ |' *tvars-match mp* $\to \mathcal{T}(C',\{x : \iota\ in\ \mathcal{V}.\ \iota \in S\})$
shows *wf-match* (*subst-match-problem-set* σ *mp*)
⟨*proof*⟩

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

lemma *wf-pat-subst*:
assumes *s*: $\sigma :_s \mathcal{V}$ |' *tvars-pat pp* $\to \mathcal{T}(C',\{x : \iota\ in\ \mathcal{V}.\ \iota \in S\})$
shows *wf-pat* (*subst-pat-problem-set* σ *pp*)
⟨*proof*⟩

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

lemma *wf-pat-iff*: *wf-pat pp* ⟷ (∀ (x,ι) ∈ *tvars-pat pp*. $\iota \in S$)
⟨*proof*⟩

The reduction of match problems preserves completeness.

lemma *mp-step-pcorrect*: *mp* $\to_s$ *mp'* ⟹ *match-complete-wrt* σ *mp* = *match-complete-wrt* σ *mp'*
⟨*proof*⟩

lemma *mp-fail-pcorrect1*:
assumes *mp-fail mp* $\sigma :_s$ *sort-annotated* |' *tvars-match mp* $\to \mathcal{T}(C,X)$
shows ¬ *match-complete-wrt* σ *mp*
⟨*proof*⟩

lemma *mp-fail-pcorrect*:
assumes *f*: *mp-fail mp* **and** *s*: $\sigma :_s \{x : \iota\ in\ \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C)$ **and** *wf*: *wf-match mp*
shows ¬ *match-complete-wrt* σ *mp*
⟨*proof*⟩

end

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

locale *pattern-completeness-context-with-assms* = *pattern-completeness-context S*

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

lemma *sorts-non-empty*: *s* ∈ *S* ⟹ ∃ *t*. *t* : *s in* $\mathcal{T}$(*C*,∅)
⟨*proof*⟩

lemma *inf-sort-not-bdd*: *s* ∈ *S* ⟹ ¬ *bdd-above* (*size* ' {*t* . *t* : *s in* $\mathcal{T}$(*C*,∅)}) ⟷ *inf-sort s*
⟨*proof*⟩

lemma *C-nth-S*: *f* : *ss* → *s in C* ⟹ *i* < *length ss* ⟹ *ss*!*i* ∈ *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'* ⟹ *wf-match mp* ⟹ *wf-match mp'*
⟨*proof*⟩

lemma *pp-step-wf*: *pp* $\Rightarrow_s$ *pp'* ⟹ *wf-pat pp* ⟹ *wf-pat pp'*
⟨*proof*⟩

theorem *P-step-set-wf*: *P* $\Rrightarrow_s$ *P'* ⟹ *wf-pats P* ⟹ *wf-pats P'*
⟨*proof*⟩

Soundness requires some preparations

definition *σg* :: *nat*×*'s* ⇒ (*'f*,*'v*) *term* **where**
σg x = (*SOME t*. *t* : *snd x in* $\mathcal{T}$(*C*,∅))

lemma *σg*: *σg* $:_s$ {*x* : *ι in sort-annotated*. *ι* ∈ *S*} → $\mathcal{T}$(*C*,∅)
⟨*proof*⟩

lemma *wf-pat-complete-iff*:
assumes *wf-pat pp*
shows *pat-complete C pp* ⟷ (∀ *σ* $:_s$ {*x* : *ι in* $\mathcal{V}$. *ι* ∈ *S*} → $\mathcal{T}$(*C*). ∃ *mp* ∈ *pp*. *match-complete-wrt σ mp*)
(**is** *?l* ⟷ *?r*)

⟨*proof*⟩

lemma *wf-pats-complete-iff*:
assumes *wf*: *wf-pats P*
shows *pats-complete C P* ⟷
$(\forall \sigma :_s \{x : \iota \text{ in } \mathcal{V}.\ \iota \in S\} \to \mathcal{T}(C).\ \forall pp \in P.\ \exists mp \in pp.\ \textit{match-complete-wrt}\ \sigma$
$mp)$
(**is** *?l* ⟷ *?r*)
⟨*proof*⟩

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

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

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

lemma *pp-step-pcorrect*:
$pp \Rightarrow_s pp' \Longrightarrow \textit{wf-pat}\ pp \Longrightarrow \textit{pat-complete}\ C\ pp = \textit{pat-complete}\ C\ pp'$
⟨*proof*⟩

lemma *pp-success-pcorrect*: $\textit{pp-success}\ pp \Longrightarrow \textit{pat-complete}\ C\ pp$
⟨*proof*⟩

theorem *P-step-set-pcorrect*:
$P \Rrightarrow_s P' \Longrightarrow \textit{wf-pats}\ P \Longrightarrow \textit{pats-complete}\ C\ P \longleftrightarrow \textit{pats-complete}\ C\ P'$
⟨*proof*⟩
end

Represent a variable-form as a set of maps.

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

definition $\textit{pat-of-var-form}\ \textit{ff} = \textit{match-of-var-form}\ `\ \textit{ff}$

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

definition $\textit{var-form-of-pat}\ pp = \textit{var-form-of-match}\ `\ pp$

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

definition *var-form-match* **where**
$\textit{var-form-match}\ mp \longleftrightarrow mp \subseteq range\ (\textit{map-prod}\ Var\ Var)$

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

lemma *match-of-var-form-of-match*:

assumes *var-form-match mp*
shows *match-of-var-form* (*var-form-of-match mp*) = *mp*
⟨*proof*⟩

lemma *tvars-match-var-form*:
assumes *var-form-match mp*
shows *tvars-match mp* = {*v*. ∃ *x*. (*Var v*, *Var x*) ∈ *mp*}
⟨*proof*⟩

lemma *pat-of-var-form-pat*:
assumes *var-form-pat pp*
shows *pat-of-var-form* (*var-form-of-pat pp*) = *pp*
⟨*proof*⟩

lemma *tvars-pat-var-form*: *tvars-pat* (*pat-of-var-form ff*) = *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*) ⟷
(∀ σ :$_s$ $\mathcal{V}$ |' *tvars-var-form-pat ff* → $\mathcal{T}(C)$. ∃ *f* ∈ *ff*. ∃ μ. ∀ *x*. ∀ *y* ∈ *f x*. σ *y* = μ *x*)
⟨*proof*⟩

lemma *pat-complete-var-form-set*:
pat-complete C (*pat-of-var-form ff*) ⟷
(∀ σ :$_s$ $\mathcal{V}$ |' *tvars-var-form-pat ff* → $\mathcal{T}(C)$. ∃ *f* ∈ *ff*. ∃ μ. ∀ *x*. σ ' *f x* ⊆ {μ *x*})
⟨*proof*⟩

lemma *pat-complete-var-form-Uniq*:
pat-complete C (*pat-of-var-form ff*) ⟷
(∀ σ :$_s$ $\mathcal{V}$ |' *tvars-var-form-pat ff* → $\mathcal{T}(C)$. ∃ *f* ∈ *ff*. ∀ *x*. *UNIQ* (σ ' *f x*))
⟨*proof*⟩

lemma *ex-var-form-pat*: (∃ *f*∈*var-form-of-pat pp*. *P f*) ⟷ (∃ *mp* ∈ *pp*. *P* (*var-form-of-match mp*))
⟨*proof*⟩

lemma *pat-complete-var-form-nat*:
assumes *fin*: ∀ (*x*,ι) ∈ *tvars-var-form-pat ff*. *finite-sort C* ι
and *uniq*: ∀ *f* ∈ *ff*. ∀ *x*::'*v*. *UNIQ* (*snd* ' *f x*)
shows *pat-complete C* (*pat-of-var-form ff*) ⟷
(∀ α. (∀ *v* ∈ *tvars-var-form-pat ff*. α *v* < *card-of-sort C* (*snd v*)) ⟶
(∃ *f* ∈ *ff*. ∀ *x*. *UNIQ* (α ' *f x*)))
(**is** *?l* ⟷ (∀ α. *?s* α ⟶ *?r* α))

⟨*proof*⟩

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

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

lemma *finite-var-form-matchD*:
assumes *finite-var-form-match C mp* **and** (t,l) ∈ *mp*
shows ∃ *x ι y*. *t* = *Var* (x,ι) ∧ *l* = *Var y* ∧ *finite-sort C ι* ∧
(∀ *z*. (*Var z*, *Var y*) ∈ *mp* ⟶ *snd z* = *ι*)
⟨*proof*⟩

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

lemma *finite-var-form-patD*:
assumes *finite-var-form-pat C pp mp* ∈ *pp* (t,l) ∈ *mp*
shows ∃ *x ι y*. *t* = *Var* (x,ι) ∧ *l* = *Var y* ∧ *finite-sort C ι* ∧
(∀ *z*. (*Var z*, *Var y*) ∈ *mp* ⟶ *snd z* = *ι*)
⟨*proof*⟩

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

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

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

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

end

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

lemma *finite-var-form-pat-pat-complete*:
  assumes *fvf*: *finite-var-form-pat C pp*
  shows *pat-complete C pp* $\longleftrightarrow$
    $(\forall \alpha. (\forall v \in \textit{tvars-pat pp}. \alpha\ v < \textit{card-of-sort}\ C\ (\textit{snd}\ v)) \longrightarrow$
    $(\exists mp \in pp. \forall x. \textit{UNIQ}\ \{\alpha\ y\ |y.\ (\textit{Var}\ y,\ \textit{Var}\ x) \in mp\}))$
⟨*proof*⟩

end

7 A Multiset-Based Inference System to Decide Pattern Completeness

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

7.1 Definition of the Inference Rules

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

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

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

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

  where *mp-mset* ≡ *set-mset*

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

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

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

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

context *pattern-completeness-context*
begin

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

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

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

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

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

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

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

inf-sort (*snd x*)
  ∧ (*improved* ⟶ *s* = *Var x* ∧ *is-Fun t*) ⟹
  add-mset mp pp $\Rightarrow_m$ *mset* (*map* (λ τ. *subst-pat-problem-mset* τ (*add-mset mp pp*)) (τ*s-list n x*))
| *pat-inf-var-conflict*: *Ball* (*pat-mset pp*) *inf-var-conflict* ⟹ *pp* ≠ {#}
  ⟹ *Ball* (*tvars-pat* (*pat-mset pp'*)) (λ *x*. ¬ *inf-sort* (*snd x*)) ⟹
  (¬ *improved* ⟹ *pp'* = {#})
  ⟹ *pp* + *pp'* $\Rightarrow_m$ {# *pp'* #}

inductive *pat-fail* :: ('*f*,'*v*,'*s*)*pat-problem-mset* ⇒ *bool* **where**
  pat-empty: *pat-fail* {#}

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

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

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

7.2 The evaluation cannot get stuck

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

lemma *pat-mset-fresh-vars*:
  ∃ *n*. *tvars-disj-pp* {*n*..<*n* + *m*} (*pat-mset p*)
⟨*proof*⟩

lemma *mp-mset-in-pat-mset*: *mp* ∈# *pp* ⟹ *mp-mset mp* ∈ *pat-mset pp*
  ⟨*proof*⟩
lemma *mp-step-mset-cong*:
  assumes $(\rightarrow_m)^{**}$ *mp mp'*
  shows (*add-mset* (*add-mset mp p*) *P*, *add-mset* (*add-mset mp' p*) *P*) ∈ $⇛^*$
  ⟨*proof*⟩

lemma *mp-step-mset-vars*: **assumes** *mp* $\rightarrow_m$ *mp'*
  shows *tvars-match* (*mp-mset mp*) ⊇ *tvars-match* (*mp-mset mp'*)
  ⟨*proof*⟩

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

shows *tvars-match* (*mp-mset mp*) ⊇ *tvars-match* (*mp-mset mp'*)
⟨*proof*⟩

end

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

lemma *pat-fail-or-trans-or-finite-var-form*:
fixes *p* :: (*'f*,*'v*,*'s*) *pat-problem-mset*
assumes *improved* $\Longrightarrow$ *infinite* (*UNIV* :: *'v set*) **and** *wf*: *wf-pat* (*pat-mset p*)
shows *pat-fail p* ∨ (∃ *ps*. *p* $\Rightarrow_m$ *ps*) ∨ (*improved* ∧ *finite-var-form-pat C* (*pat-mset p*))
⟨*proof*⟩

context
assumes *non-improved*: ¬ *improved*
begin

lemma *pat-fail-or-trans*: *wf-pat* (*pat-mset p*) $\Longrightarrow$ *pat-fail p* ∨ (∃ *ps*. *p* $\Rightarrow_m$ *ps*)
⟨*proof*⟩

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

theorem *P-step-NF*:
assumes *wf*: *wf-pats* (*pats-mset P*) **and** *NF*: *P* ∈ *NF* ⇛
shows *P* ∈ {{#}, *bottom-mset*}
⟨*proof*⟩
end

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

lemma *pat-fail-or-trans-or-fvf*:
fixes *p* :: (*'f*,*'v*,*'s*) *pat-problem-mset*
assumes *wf-pat* (*pat-mset p*)
shows *pat-fail p* ∨ (∃ *ps*. *p* $\Rightarrow_m$ *ps*) ∨ *finite-var-form-pat C* (*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*) *pats-problem-mset*) ∈ *NF* ⇛
and *p*: *p* ∈# *P*
shows *finite-var-form-pat C* (*pat-mset p*)
⟨*proof*⟩

end

end

7.3 Termination

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

fun *fun-diff* :: $('f,'v)term \Rightarrow ('f,'w)term \Rightarrow nat$ **where**
  fun-diff l (*Var x*) = *num-funs l*
| *fun-diff* (*Fun g ls*) (*Fun f ts*) = (*if f* = *g* ∧ *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*: (⋀ *y*. *y* ∈# *N* ⟹ (*y*,*x*) ∈ *R*) ⟹ (*N* + *M*, *add-mset x M*) ∈ *mult R*
  ⟨*proof*⟩

lemma *fun-diff-num-funs*: *fun-diff l t* ≤ *num-funs l*
⟨*proof*⟩

lemma *fun-diff-subst*: *fun-diff l* (*t* · σ) ≤ *fun-diff l t*
⟨*proof*⟩

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

lemma *sum-union-le-nat*: *sum* (*f* :: *'a* ⇒ *nat*) (*A* ∪ *B*) ≤ *sum f A* + *sum f B*
  ⟨*proof*⟩

lemma *sum-le-sum-list-nat*: *sum f* (*set xs*) ≤ (*sum-list* (*map f xs*) :: *nat*)
⟨*proof*⟩

lemma *bdd-above-has-Maximum-nat*: *bdd-above* (*A* :: *nat set*) ⟹ *A* ≠ {} ⟹ *has-Maximum A*
  ⟨*proof*⟩

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

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

abbreviation (*input*) *sum-ms* :: (*'a* ⇒ *nat*) ⇒ *'a multiset* ⇒ *nat* **where**
sum-ms f ms ≡ *sum-mset* (*image-mset f ms*)

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

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

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

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

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

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

abbreviation *gt-rel-pat* (**infix** ‹≻› *50*) **where**
pp ≻ *pp'* ≡ (*pp'*,*pp*) ∈ ≺

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

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

lemma *wf-rel-pat*: *wf* ≺
⟨*proof*⟩

lemma *wf-rel-pats*: *wf* ≺*mul*
⟨*proof*⟩

lemma *tvars-match-fin*:
finite (*tvars-match* (*mp-mset mp*))
⟨*proof*⟩

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

lemma *tvars-match-mono*: $mp \subseteq\# mp' \Longrightarrow$ *tvars-match* (*mp-mset mp*) $\subseteq$ *tvars-match* (*mp-mset mp'*)
⟨*proof*⟩

lemma *meas-finvars-mono*: **assumes** *tvars-match* (*mp-mset mp*) $\subseteq$ *tvars-match* (*mp-mset mp'*)
shows *meas-finvars* $\{\#mp\#\} \leq$ *meas-finvars* $\{\#mp'\#\}$
⟨*proof*⟩

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

lemma *rel-mp-mp-step-mset*:
fixes *mp* :: ('*f*,'*v*,'*s*) *match-problem-mset*
assumes $mp \to_m mp'$
shows $\{\#mp\#\} \succ \{\#mp'\#\}$
⟨*proof*⟩

lemma *sum-ms-image*: *sum-ms f* (*image-mset g ms*) = *sum-ms* (*f o g*) *ms*
⟨*proof*⟩

lemma *meas-diff-subst-le*: *meas-diff* (*subst-pat-problem-mset* τ *p*) $\leq$ *meas-diff p*
⟨*proof*⟩

lemma *meas-sub*: **assumes** *sub*: $p' \subseteq\# p$
shows *meas-diff* $p' \leq$ *meas-diff p*
meas-finvars $p' \leq$ *meas-finvars p*
meas-symbols $p' \leq$ *meas-symbols p*
⟨*proof*⟩

lemma *meas-sub-rel-pat*: **assumes** *sub*: $p' \subset\# p$
shows $p \succ p'$
⟨*proof*⟩

lemma *max-size-term-of-sort*: **assumes** *sS*: $s \in S$ **and** *inf*: $\neg$ *inf-sort s*
shows $\exists$ *t*. *t* : *s in* $\mathcal{T}(C)$ $\wedge$ *max-size s* = *size t* $\wedge$ ($\forall$ *t'*. *t'* : *s in* $\mathcal{T}(C)$ $\longrightarrow$ *size t'* $\leq$ *size t*)
⟨*proof*⟩

lemma *max-size-max*: **assumes** *sS*: $s \in S$
and *inf*: $\neg$ *inf-sort s*
and *sort*: *t* : *s in* $\mathcal{T}(C)$
shows *size t* $\leq$ *max-size s*
⟨*proof*⟩

lemma *finite-sort-size*: **assumes** *c*: *c* : *map snd vs* $\to$ *s in C*
and *inf*: $\neg$ *inf-sort s*
shows *sum* (*max-size o snd*) (*set vs*) < *max-size s*
⟨*proof*⟩

lemma *rel-pp-step-mset*:
  fixes p :: $('f,'v,'s)$ *pat-problem-mset*
  assumes $p \Rightarrow_m ps$
  and $p' \in\# ps$
shows $p \succ p'$
  ⟨*proof*⟩

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

lemma *rel-P-trans*:
  assumes $P \Rrightarrow_m P'$
  shows $P \succ$*mul* P'
  ⟨*proof*⟩

termination of the multiset based implementation

theorem *SN-P-step*: *SN* $\Rrightarrow$
⟨*proof*⟩

7.4 Partial Correctness via Refinement

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

lemma *mp-step-cong*: $mp1 \rightarrow_s mp2 \Longrightarrow mp1 = mp1' \Longrightarrow mp2 = mp2' \Longrightarrow mp1' \rightarrow_s mp2'$ ⟨*proof*⟩

lemma *mp-step-mset-mp-trans*: $mp \rightarrow_m mp' \Longrightarrow$ *mp-mset* $mp \rightarrow_s$ *mp-mset* mp'
⟨*proof*⟩

lemma *mp-fail-cong*: *mp-fail* $mp \Longrightarrow mp = mp' \Longrightarrow$ *mp-fail* mp' ⟨*proof*⟩

lemma *match-fail-mp-fail*: *match-fail* $mp \Longrightarrow$ *mp-fail* (*mp-mset* mp)
⟨*proof*⟩

lemma *P-step-set-cong*: $P \Rrightarrow_s Q \Longrightarrow P = P' \Longrightarrow Q = Q' \Longrightarrow P' \Rrightarrow_s Q'$ ⟨*proof*⟩

lemma *P-step-mset-imp-set*: **assumes** $P \Rrightarrow_m Q$
  shows *pats-mset* $P \Rrightarrow_s$ *pats-mset* Q
  ⟨*proof*⟩

lemma *P-step-pp-trans*: **assumes** $(P,Q) \in \Rrightarrow$
  shows *pats-mset* $P \Rrightarrow_s$ *pats-mset* Q
  ⟨*proof*⟩

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

corollary *P-steps-pcorrect*: **assumes** *wf*: *wf-pats* (*pats-mset P*)
and *step*: $(P,Q) \in \Rrightarrow^*$
shows *wf-pats* (*pats-mset Q*) ∧ (*pats-complete C* (*pats-mset P*) ⟷ *pats-complete C* (*pats-mset Q*))
⟨*proof*⟩

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

theorem *P-step*:
assumes *non-improved*: ¬ *improved*
and *wf*: *wf-pats* (*pats-mset P*) **and** *NF*: $(P,Q) \in \Rrightarrow^!$
shows *Q* = {#} ∧ *pats-complete C* (*pats-mset P*) — either the result is and input P is complete
∨ *Q* = *bottom-mset* ∧ ¬ *pats-complete C* (*pats-mset P*) — or the result = bot and P is not complete
⟨*proof*⟩

theorem *P-step-improved*:
fixes *P* :: ('*f*,'*v*,'*s*) *pats-problem-mset*
assumes *improved*
and *inf*: *infinite* (*UNIV* :: '*v set*)
and *wf*: *wf-pats* (*pats-mset P*) **and** *NF*: $(P,Q) \in \Rrightarrow^!$
shows *pats-complete C* (*pats-mset P*) ⟷ *pats-complete C* (*pats-mset Q*) — equivalence
p ∈# *Q* ⟹ *finite-var-form-pat C* (*pat-mset p*) — all remaining problems are in finite-var-form
⟨*proof*⟩

end
end

8 A List-Based Implementation to Decide Pattern Completeness

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

8.1 Definition of Algorithm

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  where *pats-mset-list* $\equiv$ *mset o map pat-mset-list*

definition *subst-match-problem-list* :: $('f,nat \times 's)$*subst* $\Rightarrow$ $('f,'v,'s)$*match-problem-list*
$\Rightarrow$ $('f,'v,'s)$*match-problem-list* **where**
  subst-match-problem-list τ = *map* (*subst-left* τ)

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

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

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

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

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

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

Transforming finite variable forms:

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

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

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

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

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

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

lemma *size-zip*[*termination-simp*]: *length ts* = *length ls* ⟹ *size-list* (λ*p*. *size* (*snd p*)) (*zip ts ls*)
< *Suc* (*size-list size ls*)
⟨*proof*⟩

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

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

| *Some mp′ ⇒ if mp′ = [] then None*
else pat-inner-lin-impl p (mp′ # pd))

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

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

lemma *set-pairs-of-list*: *set (pairs-of-list xs) = { (xs ! i, xs ! (Suc i)) | i. Suc i < length xs}*
⟨*proof*⟩

lemma *diff-pairs-of-list*: (∃ *x* ∈ *set xs*. ∃ *y* ∈ *set xs*. *f x* ≠ *f y*) ⟷
(∃ (*x*,*y*) ∈ *set* (*pairs-of-list xs*). *f x* ≠ *f y*) (**is** *?l* = *?r*)
⟨*proof*⟩

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

context *pattern-completeness-context*
begin

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

definition *insert-rx* :: (′*f*,*nat* × ′*s*)*term* ⇒ ′*v* ⇒ (′*f*,′*v*,′*s*)*match-problem-rx* ⇒ (′*f*,′*v*,′*s*)*match-problem-rx option* **where**
insert-rx t x rxb = (case rxb of (rx,b) ⇒ (case map-of rx x of
None ⇒ Some (((x,[t]) # rx, b))
| *Some ts ⇒ (case those (map (conflicts t) ts)*
of None ⇒ None — clash
| *Some cs ⇒ if [] ∈ set cs then Some rxb* — empty conflict means (t,x) was already part of rxb
else Some ((AList.update x (t # ts) rx, b ∨ (∃ *y* ∈ *set* (*concat cs*). *inf-sort* (*snd y*))))
)))

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

fun *decomp-impl* :: (′*f*,′*v*,′*s*)*match-problem-list* ⇒ (′*f*,′*v*,′*s*)*match-problem-lr option* **where**
decomp-impl [] = Some ([],([],False))
| *decomp-impl ((Fun f ts, Fun g ls) # mp) = (if (f,length ts) = (g,length ls) then*
decomp-impl (zip ts ls @ mp) else None)
| *decomp-impl ((Var x, Fun g ls) # mp) = (case decomp-impl mp of Some (lx,rx)* ⇒ *Some ((x,Fun g ls) # lx,rx)*
| *None ⇒ None)*

| *decomp-impl* ((*t*, *Var y*) # *mp*) = (*case decomp-impl mp of Some* (*lx*,*rx*) ⇒
(*case insert-rx t y rx of Some rx′* ⇒ *Some* (*lx*,*rx′*) | *None* ⇒ *None*)
| *None* ⇒ *None*)

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

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

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

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

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

partial-function (*tailrec*) *decomp′-main-loop* **where**
decomp′-main-loop n xs list out = (*case list of*
[] ⇒ (*n*, *out*) — one might change to (rev out) in order to preserve the order
| ((*x*,*ts*) # *rxs*) ⇒ (*if tl ts* = [] ∨ (∃ *t* ∈ *set ts. is-Var t*) ∨ *x* ∈ *set xs*
then decomp′-main-loop n xs rxs ((*x*,*ts*) # *out*)
else let l = *length* (*args* (*hd ts*));
fresh = *map renNat* [*n* ..< *n* + *l*];
new = *zipAll fresh* (*map args ts*);
cleaned = *filter* (λ (*y*,*ts′*). *tl ts′* ≠ []) (*map* (λ (*y*,*ts′*). (*y*, *remdups ts′*)) *new*)
in decomp′-main-loop (*n* + *l*) *xs* (*cleaned* @ *rxs*) *out*))

definition *decomp′-impl* **where**

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

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

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

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

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

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

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

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

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

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

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

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

8.2 Partial Correctness of the Implementation

TODO: move

lemma *mset-sum-reindex*: (∑ *x*∈#*A*. *image-mset* (*f x*) *B*) = (∑ *i*∈#*B*. {#*f x i*. *x* ∈# *A*#})
⟨*proof*⟩

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* ∈ *set bss* ⟹ *length bs* = *n*
shows *zipAll as bss* = *map* (λ *i*. (*as* ! *i*, *map* (λ *bs*. *bs* ! *i*) *bss*)) [*0*..<*n*]
⟨*proof*⟩

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

lemma *mset-concat-union*:
mset (*concat xs*) = $\sum_{\#}$ (*mset* (*map mset xs*))
⟨*proof*⟩

lemma *in-map-mset*[*intro*]:
a ∈# *A* ⟹ *f a* ∈# *image-mset f A*
⟨*proof*⟩

lemma *mset-update*: *map-of xs x* = *Some y* ⟹
mset (*AList.update x z xs*) = (*mset xs* − {# (*x*,*y*) #}) + {# (*x*,*z*) #}
⟨*proof*⟩

lemma *set-update*: *map-of xs x* = *Some y* ⟹ *distinct* (*map fst xs*) ⟹
set (*AList.update x z xs*) = *insert* (*x*,*z*) (*set xs* − {(*x*,*y*)})
⟨*proof*⟩

lemma *mp-rx-append*: *mp-rx* (*xs* @ *ys*, *b*) = *mp-rx* (*xs*,*b*) + *mp-rx* (*ys*,*b*)
⟨*proof*⟩

lemma *mp-rx-Cons*: *mp-rx* (*p* # *xs*, *b*) = *mp-list* (*case p of* (*x*, *ts*) ⇒ *map* (λ*t*. (*t*, *Var x*)) *ts*)
+ *mp-rx* (*xs*,*b*)
⟨*proof*⟩

lemma *set-tvars-match-list*: *set* (*tvars-match-list mp*) = *tvars-match* (*set mp*)
⟨*proof*⟩

lemma *set-tvars-pat-list*: *set* (*tvars-pat-list pp*) = *tvars-pat* (*pat-list pp*)
⟨*proof*⟩

lemma *finite-var-form-pat-pat-complete-list*:
fixes *pp*::(′*f*,′*v*,′*s*) *pat-problem-list* **and** *C*
assumes *fvf*: *finite-var-form-pat C* (*pat-list pp*)
and *pp*: *pp* = *pat-of-var-form-list fvf*
and *dist*: *Ball* (*set fvf*) (*distinct o map fst*)
shows *pat-complete C* (*pat-list pp*) ⟷
(∀ α. (∀ *v* ∈ *set* (*tvars-pat-list pp*). α *v* < *card-of-sort C* (*snd v*)) ⟶
(∃ *c* ∈ *set* (*map* (*map snd*) *fvf*).
∀ *vs* ∈ *set c*. *UNIQ* (α ‘ *set vs*)))
⟨*proof*⟩

lemma *pat-complete-via-cnf*:
assumes *fvf*: *finite-var-form-pat C* (*pat-list pp*)
and *pp*: *pp* = *pat-of-var-form-list fvf*
and *dist*: *Ball* (*set fvf*) (*distinct o map fst*)
and *cnf*: *cnf* = *map* (*map snd*) *fvf*
shows *pat-complete C* (*pat-list pp*) ⟷
(∀ α. (∀ v ∈ *set* (*concat* (*concat cnf*)). α v < *card-of-sort C* (*snd v*)) ⟶
(∃ c ∈ *set cnf*. ∀ *vs* ∈ *set c*. *UNIQ* (α ‘ *set vs*)))
⟨*proof*⟩

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

Various well-formed predicates for intermediate results

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

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

definition *wf-ts3* :: (′*f*, *nat* × ′*s*) *term list* ⇒ *bool* **where**
wf-ts3 ts = (∃ *t* ∈ *set ts*. *is-Var t*)

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

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

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

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

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

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

definition *wf-lr3* :: (′*f*,′*v*,′*s*)*match-problem-lr* ⇒ *bool*

where *wf-lr3 pair* = (*case pair of* (*lx*,*rx*) ⇒ *wf-lx lx* ∧ (*if lx* = [] *then wf-rx3 rx else wf-rx rx*))

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

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

lemma *wf-rx-mset*: **assumes** *mset rx* = *mset rx'*
shows *wf-rx* (*rx*,*b*) = *wf-rx* (*rx'*,*b*)
⟨*proof*⟩

lemma *wf-rx2-mset*: **assumes** *mset rx* = *mset rx'*
shows *wf-rx2* (*rx*,*b*) = *wf-rx2* (*rx'*,*b*)
⟨*proof*⟩

lemma *wf-lr2-mset*: **assumes** *mset rx* = *mset rx'*
shows *wf-lr2* (*lx*,(*rx*,*b*)) = *wf-lr2* (*lx*,(*rx'*,*b*))
⟨*proof*⟩

lemma *mp-lr-mset*: **assumes** *mset rx* = *mset rx'*
shows *mp-lr* (*lx*,(*rx*,*b*)) = *mp-lr* (*lx*,(*rx'*,*b*))
⟨*proof*⟩

lemma *mp-list-lr*: *mp-list* (*mp-lr-list mp*) = *mp-lr mp*
⟨*proof*⟩

lemma *pat-mset-list-lr*: *pat-mset-list* (*map mp-lr-list pp*) = *pat-lr pp*
⟨*proof*⟩

lemma *size-term-0*[*simp*]: *size* (*t* :: ('*f*,'*v*)*term*) > *0*
⟨*proof*⟩

lemma *wf-ts-no-conflict-alt-def*: (∀ *j* < *length ts*. ∀ *i* < *j*. *conflicts* (*ts* ! *i*) (*ts* ! *j*) ≠ *None*)
⟷ (∀ *s t*. *s* ∈ *set ts* ⟶ *t* ∈ *set ts* ⟶ *conflicts s t* ≠ *None*) (**is** *?l* = *?r*)
⟨*proof*⟩

Continue with properties of the sub-algorithms

lemma *insert-rx*: **assumes** *res*: *insert-rx t x rxb* = *res*
and *wf*: *wf-rx rxb*
and *mp*: *mp* = (*ls*,*rxb*)
shows *res* = *Some rx'* ⟹ $(\rightarrow_m)^{**}$ (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*)) (*mp-lr* (*ls*,*rx'*) + *M*) ∧ *wf-rx rx'*

$\wedge$ *lvars-mp* (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*)) $\supseteq$ *lvars-mp* (*mp-lr* (*ls*,*rx'*) + *M*)
res = *None* $\Longrightarrow$ *match-fail* (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*))
⟨*proof*⟩

lemma *decomp-impl*: *decomp-impl mp* = *res* $\Longrightarrow$
(*res* = *Some mp'* $\longrightarrow$ $(\rightarrow_m)^{**}$ (*mp-list mp* + *M*) (*mp-lr mp'* + *M*) $\wedge$ *wf-lr mp'*

$\wedge$ *lvars-mp* (*mp-list mp* + *M*) $\supseteq$ *lvars-mp* (*mp-lr mp'* + *M*))
$\wedge$ (*res* = *None* $\longrightarrow$ ($\exists$ *mp'*. $(\rightarrow_m)^{**}$ (*mp-list mp* + *M*) *mp'* $\wedge$ *match-fail mp'*))
⟨*proof*⟩

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

lemma *pat-inner-lin-impl*: **assumes** *pat-inner-lin-impl p pd* = *res*
and *wf-pat-lx pd* $\forall$ *mp* $\in$ *set p*. *ll-mp* (*mp-list mp*)
and *tvars-pat* (*pat-mset* (*pat-mset-list p* + *pat-lx pd*)) $\subseteq$ *V*
shows *res* = *None* $\Longrightarrow$ (*add-mset* (*pat-mset-list p* + *pat-lx pd*) *P*, *P*) $\in$ $\Rrightarrow^+$
and *res* = *Some p'* $\Longrightarrow$ (*add-mset* (*pat-mset-list p* + *pat-lx pd*) *P*, *add-mset* (*pat-lx p'*) *P*) $\in$ $\Rrightarrow^*$
$\wedge$ *wf-pat-lx p'* $\wedge$ *tvars-pat* (*pat-mset* (*pat-lx p'*)) $\subseteq$ *V*
⟨*proof*⟩

lemma *pat-mset-list*: *pat-mset* (*pat-mset-list p*) = *pat-list p*
⟨*proof*⟩

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

= *vars-mp-mset* (*mp-list mp*)
⟨*proof*⟩

lemma *subst-conversion*: *map* ($\lambda\tau$. *subst-pat-problem-mset* τ (*pat-mset-list p*)) *xs* =
map pat-mset-list (*map* ($\lambda\tau$. *subst-pat-problem-list* τ *p*) *xs*)
⟨*proof*⟩

lemma *ll-mp-subst*: *ll-mp* (*mp-list* (*subst-match-problem-list* τ *mp*)) = *ll-mp* (*mp-list mp*)
⟨*proof*⟩

lemma *ll-pp-subst*: *ll-pp* (*subst-pat-problem-list* τ *p*) = *ll-pp p*

⟨*proof*⟩

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

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

lemma *pats-mset-list*: *pats-mset* (*pats-mset-list ps*) = *pat-list* ' *set ps*
⟨*proof*⟩

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

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

lemma *match-var-impl*: **assumes** *wf*: *wf-lr mp*
and *match-var-impl mp* = (*xs*,*mpFin*)
shows $(\rightarrow_m)^{**}$ (*mp-lr mp*) (*mp-lr mpFin*)
and *wf-lr2 mpFin*
and *lvars-mp* (*mp-lr mp*) $\supseteq$ *lvars-mp* (*mp-lr mpFin*)
and *set xs* = *lvars-mp* (*mp-list* (*mp-lx* (*fst mpFin*)))
⟨*proof*⟩

lemma *match-steps-impl*: **assumes** *match-steps-impl mp = res*
shows *res = Some* (*xs*,*mp'*) $\Longrightarrow (\rightarrow_m)^{**}$ (*mp-list mp*) (*mp-lr mp'*) $\wedge$ *wf-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* $\Longrightarrow \exists$ *mp'*. $(\rightarrow_m)^{**}$ (*mp-list mp*) *mp'* $\wedge$ *match-fail mp'*
⟨*proof*⟩

lemma *finite-sort-imp-finite-sort-vars*:

assumes *t* : *σ* *in* $\mathcal{T}(C,\mathcal{V})$
and *x* ∈ *vars t*
and ¬ *inf-sort σ*
shows ¬ *inf-sort* (*snd x*)
⟨*proof*⟩

context
fixes *CC* :: *'f* × *'s list* ⇒ *'s option*
and *renVar* :: *'v* ⇒ *'v*
and *renNat* :: *nat* ⇒ *'v*
and *fidl-solver* :: ((*nat*×*'s*) × *int*)*list* × - ⇒ *bool*
assumes *CC*: *improved* ⟹ *CC* = *C*
and *renaming-ass*: *improved* ⟹ *renaming-funs renNat renVar*
and *fidl-solver*: *improved* ⟹ *finite-idl-solver fidl-solver*
begin

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

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

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

lemma *lvar-cond-simps*[*simp*]:
lvar-cond n (*insert x A*) = (*x* ∈ *allowed-vars n* ∧ *lvar-cond n A*)
lvar-cond n {}
lvar-cond n (*A* ∪ *B*) = (*lvar-cond n A* ∧ *lvar-cond n B*)
lvar-cond n (⋃ *As*) = (∀ *A* ∈ *As*. *lvar-cond n A*)
⟨*proof*⟩

lemma *lvar-cond-mono*: *n* ≤ *n'* ⟹ *lvar-cond n V* ⟹ *lvar-cond n' V*
⟨*proof*⟩

lemma *pair-fst-imageI*: (*a*,*b*) ∈ *c* ⟹ *a* ∈ *fst* ‘ *c* ⟨*proof*⟩

lemma *not-in-fstD*: *x* ∉ *fst* ‘ *a* ⟹ ∀ *z*. (*x*,*z*) ∉ *a* ⟨*proof*⟩

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

lemma *many-match-steps*:
assumes $\bigwedge$ *t l*. (*t*,*l*) $\in$# *mp1* $\Longrightarrow$ $\exists$ *x*. *l* = *Var x* $\wedge$ *x* $\notin$ *lvars-mp* (*mp1* − {# (*t*,*l*) #} + *mp2*)
shows $(\rightarrow_m)^{**}$ (*mp1* + *mp2*) *mp2*
⟨*proof*⟩

lemma *decomp′-impl*: **assumes**
wf-lr2 mp
set xs = *lvars-mp* (*mp-list* (*mp-lx* (*fst mp*)))
lvar-cond-mp n (*mp-lr mp*)
Decomp′-impl n xs mp = (*n′*,*mp′*)
improved
shows *wf-lr3 mp′*
lvar-cond-mp n′ (*mp-lr mp′*)
$(\rightarrow_m)^{**}$ (*mp-lr mp*) (*mp-lr mp′*)
$n \leq n'$
⟨*proof*⟩

lemma *match-decomp′-impl*: **assumes** *Match-decomp′-impl n mp* = *res*
and *lvc*: *lvar-cond-mp n* (*mp-list mp*)
shows *res* = *Some* (*n′*,*mp′*) $\Longrightarrow$ $(\rightarrow_m)^{**}$ (*mp-list mp*) (*mp-lr mp′*) $\wedge$ *wf-lr3 mp′* $\wedge$ *lvar-cond-mp n′* (*mp-lr mp′*) $\wedge$ $n \leq n'$
and *res* = *None* $\Longrightarrow$ $\exists$ *mp′*. $(\rightarrow_m)^{**}$ (*mp-list mp*) *mp′* $\wedge$ *match-fail mp′*
⟨*proof*⟩

lemma *pat-inner-impl*: **assumes** *Pat-inner-impl n p pd* = *res*
and *wf-pat-lr pd*
and *tvars-pat* (*pat-mset* (*pat-mset-list p* + *pat-lr pd*)) $\subseteq$ *V*
and *lvar-cond-pp n* (*pat-mset-list p* + *pat-lr pd*)
shows *res* = *None* $\Longrightarrow$ (*add-mset* (*pat-mset-list p* + *pat-lr pd*) *P*, *P*) $\in \Rrightarrow^+$
and *res* = *Some* (*n′*,*p′*) $\Longrightarrow$ (*add-mset* (*pat-mset-list p* + *pat-lr pd*) *P*, *add-mset* (*pat-lr p′*) *P*) $\in \Rrightarrow^*$
$\wedge$ *wf-pat-lr p′* $\wedge$ *tvars-pat* (*pat-mset* (*pat-lr p′*)) $\subseteq$ *V* $\wedge$ *lvar-cond-pp n′* (*pat-lr p′*) $\wedge$ $n \leq n'$
⟨*proof*⟩

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

lemma *pat-impl*:
assumes *Pat-impl n nl p* = *res*
and *vars*: *tvars-pat* (*pat-list p*) $\subseteq$ {..<*n*} $\times$ *S*
and *lvarsAll*: $\forall$ *pp* $\in$# *add-mset* (*pat-mset-list p*) *P*. *lvar-cond-pp nl pp*

shows *res* = *Incomplete* ⟹ ∃ *p'*. (*add-mset* (*pat-mset-list p*) *P*, *add-mset p' P*) ∈ ⇛* ∧ *pat-fail p'*
and *res* = *New-Problems* (*n'*,*nl'*,*ps*) ⟹ (*add-mset* (*pat-mset-list p*) *P*, *mset* (*map pat-mset-list ps*) + *P*) ∈ ⇛⁺
∧ *tvars-pat* (⋃ (*pat-list* ' *set ps*)) ⊆ {..< *n'*} × *S*
∧ (∀ *pp* ∈# *mset* (*map pat-mset-list ps*) + *P*. *lvar-cond-pp nl' pp*) ∧ *n* ≤ *n'*
and *res* = *Fin-Var-Form fvf* ⟹ *improved*
∧ (*add-mset* (*pat-mset-list p*) *P*, *add-mset* (*pat-mset-list* (*pat-of-var-form-list fvf*)) *P*) ∈ ⇛*
∧ *finite-var-form-pat C* (*pat-list* (*pat-of-var-form-list fvf*))
∧ *Ball* (*set fvf*) (*distinct o map fst*)
∧ *Ball* (*set* (*concat fvf*)) (*distinct* ∘ *snd*)
⟨*proof*⟩

lemma *non-uniq-image-diff*: ¬ *UNIQ* (*α* ' *set vs*) ⟷ (∃ *v* ∈ *set vs*. ∃ *w* ∈ *set vs*. *α v* ≠ *α w*)
⟨*proof*⟩

lemma *pat-complete-via-idl-solver*:
assumes *impr*: *improved*
and *fvf*: *finite-var-form-pat C* (*pat-list pp*)
and *wf*: *wf-pat* (*pat-list pp*)
and *pp*: *pp* = *pat-of-var-form-list fvf*
and *dist*: *Ball* (*set fvf*) (*distinct o map fst*)
and *dist2*: *Ball* (*set* (*concat fvf*)) (*distinct o snd*)
and *cnf*: *cnf* = *map* (*map snd*) *fvf*
shows *pat-complete C* (*pat-list pp*) ⟷ ¬ *fidl-solver* (*bounds-list* (*cd-sort* ∘ *snd*) *cnf*, *dist-pairs-list cnf*)
⟨*proof*⟩

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

lemma *pats-impl*: **assumes** ∀ *p* ∈ *set ps*. *tvars-pat* (*pat-list p*) ⊆ {..<*n*} × *S*
and *Ball* (*set ps*) (*λ pp*. *lvar-cond-pp nl* (*pat-mset-list pp*))
and ∀ *pp* ∈ *pat-list* ' *set ps*. *wf-pat pp*
shows *Pats-impl n nl ps* = *pats-complete C* (*pat-list* ' *set ps*)
⟨*proof*⟩

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

corollary *pat-complete-impl*:
assumes *wf*: *snd* ' ⋃ (*vars* ' *fst* ' *set* (*concat* (*concat P*))) ⊆ *S*
shows *Pat-complete-impl* (*P* :: ('*f*,'*v*,'*s*)*pats-problem-list*) ⟷ *pats-complete C* (*pat-list* ' *set P*)
⟨*proof*⟩
end

end

8.3 Getting the result outside the locale with assumptions

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

lemma *hastype-in-map-of*: *distinct* (*map fst l*) $\Longrightarrow$ *x* : σ *in map-of l* $\longleftrightarrow$ (*x*,σ) $\in$ *set l*
⟨*proof*⟩

lemma *fun-hastype-in-map-of*: *distinct* (*map fst l*) $\Longrightarrow$
x : σs $\to$ τ *in map-of l* $\longleftrightarrow$ ((*x*,σs),τ) $\in$ *set l*
⟨*proof*⟩

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

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

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

lemma *sorts-of-ssig-list*:
assumes ((*f*,σs),τ) $\in$ *set Cs*
shows *set* σs $\subseteq$ *set* (*sorts-of-ssig-list Cs*) τ $\in$ *set* (*sorts-of-ssig-list Cs*)
⟨*proof*⟩

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

lemma *max-arity-list*:
((*f*,σs),τ) $\in$ *set Cs* $\Longrightarrow$ *length* σs $\leq$ *max-arity-list Cs*
⟨*proof*⟩

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

lemma *nonempty-sort*: $\bigwedge$ σ. σ $\in$ *set* (*sorts-of-ssig-list Cs*) $\Longrightarrow$ ¬ *empty-sort* (*map-of Cs*) σ
⟨*proof*⟩

lemma *compute-inf-sorts*: σ $\in$ *compute-inf-sorts Cs* $\longleftrightarrow$ ¬ *finite-sort* (*map-of Cs*) σ
⟨*proof*⟩

lemma *compute-card-sorts*: *snd* (*compute-inf-card-sorts Cs*) = *card-of-sort* (*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*
λ s. s ∈ *compute-inf-sorts Cs*
snd (*compute-inf-card-sorts 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* :: *nat* — upper bound on arities of constructors
and *Cl* :: *'s* ⇒ (*'f* × *'s list*)*list* — a function to compute all constructors of given sort as list
and *Is* :: *'s* ⇒ *bool* — a function to indicate whether a sort is infinite
and *Cd* :: *'s* ⇒ *nat* — a function to compute finite cardinality of sort
begin

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

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

lemma *find-var-old*: *find-var False p* = *Some* (*find-var-old p*)
⟨*proof*⟩

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

folded pat-complete-impl-old-def pats-impl-old-def,
unfolded if-False Let-def]

private lemma *triv-ident*: *False* ∧ *x* ⟷ *False True* ∧ *x* ⟷ *x* ⟨*proof*⟩

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

lemma *pats-impl-old-code*[*code*]:
pats-impl-old n nl ps =
(*case ps of* [] ⇒ *True*
| *p # ps1* ⇒
(*case pat-impl-old n nl p of Incomplete* ⇒ *False*
| *New-Problems* (*n*′, *nl*′, *ps2*) ⇒ *pats-impl-old n*′ *nl*′ (*ps2* @ *ps1*)))
⟨*proof*⟩

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

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

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

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

folded pat-complete-impl-new-def pats-impl-new-def,
unfolded if-True Let-def]

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

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

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

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

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

end
end

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

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

theorem *decide-pat-complete-lin*:
assumes *dist*: *distinct* (*map fst Cs*)
and *non-empty-sorts*: *decide-nonempty-sorts* (*sorts-of-ssig-list Cs*) *Cs* = *None*
and *P*: *snd* ‘ ⋃ (*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*:
 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 Cs P* = *pats-complete* (*map-of Cs*) (*pat-list* ‘ *set P*)
⟨*proof*⟩

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

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

theorem *decide-pat-complete-fidl*:
 assumes *dist*: *distinct* (*map fst Cs*)
  and *non-empty-sorts*: *decide-nonempty-sorts* (*sorts-of-ssig-list Cs*) *Cs* = *None*
  and *P*: *snd* ‘ ⋃ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat P*))) ⊆ *set* (*sorts-of-ssig-list Cs*)
  and *ren*: *renaming-funs rn rv*
  and *fidl-solver*: *finite-idl-solver fidl-solver*
 shows *decide-pat-complete-fidl rn rv fidl-solver Cs P* ⟷ *pats-complete* (*map-of Cs*) (*pat-list* ‘ *set P*)
  (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

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

end

9 Pattern-Completeness and Related Properties

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

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

A pattern completeness decision procedure for a set of lhss

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

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

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

lemma *matches-subst*: $l\ matches\ t \Longrightarrow l\ matches\ t{\cdot}\sigma$
  ⟨*proof*⟩

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

lemma *pat-complete-lhssD*:
  assumes *comp*: $pat\text{-}complete\text{-}lhss\ C\ D\ L$ **and** *t*: $t \in \mathcal{B}(C,D,\emptyset)$
  shows $\exists\, l \in L.\ l\ matches\ t$
⟨*proof*⟩

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

definition *check-signatures* :: $(('f \times 's\ list) \times 's)list \Rightarrow (('f \times 's\ list) \times 's)list \Rightarrow$ *showsl check* **where**
  check-signatures C D = *do* {
    check (*distinct* (*map fst C*)) (*showsl-lit* (*STR ''constructor information contains duplicate''*));

check (*distinct* (*map fst D*)) (*showsl-lit* (*STR* ''*defined symbol information contains duplicate*''));
let S = *sorts-of-ssig-list C*;
check-allm (λ ((*f*,*ss*),-). *check-allm* (λ *s*. *check* (*s* ∈ *set S*)
(*showsl-lit* (*STR* ''*a defined symbol has argument sort that is not known in constructors*''))) *ss*) *D*;
(*case* (*decide-nonempty-sorts S C*) *of None* ⇒ *return* () | *Some s* ⇒ *error* (*showsl-lit* (*STR* ''*some sort is empty*'')))
}

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

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

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

lemma *pats-of-lhss-vars*: **assumes** *condD*: ∀ *x*∈*set D*. ∀ *a b*. (∀ *x2*. *x* ≠ ((*a*, *b*), *x2*)) ∨ (∀ *x*∈*set b*. *x* ∈ *S*)
shows *snd* ‘ ⋃ (*vars* ‘ *fst* ‘ *set* (*concat* (*concat* (*pats-of-lhss D lhss*)))) ⊆ *S*
⟨*proof*⟩

lemma *check-signatures*: **assumes** *isOK*(*check-signatures C D*)
shows *distinct* (*map fst C*) (**is** *?G1*)
and *distinct* (*map fst D*) (**is** *?G2*)
and ∀ *x*∈*set D*. ∀ *a b*. (∀ *x2*. *x* ≠ ((*a*, *b*), *x2*)) ∨ (∀ *x*∈*set b*. *x* ∈ *set* (*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*)) =
(∀ *t*∈$\mathcal{B}$(*map-of C*,*map-of D*). ∃ *l*∈*set lhss*. *l matches t*)
⟨*proof*⟩

theorem *decide-pat-complete-lhss*:
fixes *C D* :: ((′*f* × ′*s list*) × ′*s*) *list* **and** *lhss* :: (′*f*,′*v*)*term list*
assumes *decide-pat-complete-lhss C D lhss* = *return b*
shows *b* = *pat-complete-lhss* (*map-of C*) (*map-of D*) (*set lhss*)
⟨*proof*⟩

theorem *decide-pat-complete-linear-lhss*:
fixes *C D* :: ((′*f* × ′*s list*) × ′*s*) *list* **and** *lhss* :: (′*f*,′*v*)*term list*
assumes *decide-pat-complete-linear-lhss C D lhss* = *return b*
and *linear*: *Ball* (*set lhss*) *linear-term*
shows *b* = *pat-complete-lhss* (*map-of C*) (*map-of D*) (*set lhss*)
⟨*proof*⟩

theorem *decide-pat-complete-lhss-fidl*:
fixes *C D* :: ((′*f* × ′*s list*) × ′*s*) *list* **and** *lhss* :: (′*f*,′*v*)*term list*
assumes *decide-pat-complete-lhss-fidl rn rv fidl-solver C D lhss* = *return b*
and *ren*: *renaming-funs rn rv*
and *idl*: *finite-idl-solver fidl-solver*
shows *b* = *pat-complete-lhss* (*map-of C*) (*map-of D*) (*set lhss*)
⟨*proof*⟩

Definition of strong quasi-reducibility and a corresponding decision procedure

definition *strong-quasi-reducible* :: (′*f*,′*s*)*ssig* ⇒ (′*f*,′*s*)*ssig* ⇒ (′*f*,′*v*)*term set* ⇒ *bool* **where**
strong-quasi-reducible C D L =
(∀ *t* ∈ $\mathcal{B}$(*C*,*D*,∅::*unit*⇀′*s*). ∃ *ti* ∈ *set* (*t* # *args t*). ∃ *l* ∈ *L*. *l matches ti*)

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

definition *decide-strong-quasi-reducible* ::
((′*f* × ′*s list*) × ′*s*)*list* ⇒ ((′*f* × ′*s list*) × ′*s*)*list* ⇒ (′*f*,′*v*)*term list* ⇒ *showsl* + *bool* **where**
decide-strong-quasi-reducible C D lhss = *do* {
check-signatures C D;
let pats = *map* (λ ((*f*,*ss*),*s*). *term-and-args f* (*map Var* (*zip* [*0*..<*length ss*] *ss*))) *D*;
let P = *map* (*List.maps* (λ *pat*. *map* (λ *lhs*. [(*pat*,*lhs*)]) *lhss*)) *pats*;
return (*decide-pat-complete C P*)
}

lemma *decide-strong-quasi-reducible*:
fixes *C D* :: ((′*f* × ′*s list*) × ′*s*) *list* **and** *lhss* :: (′*f*,′*v*)*term list*

assumes *decide-strong-quasi-reducible C D lhss = return b*
shows *b = strong-quasi-reducible (map-of C) (map-of D) (set lhss)*
⟨*proof*⟩

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

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

lemma *pat-complete-imp-strong-quasi-reducible*:
pat-complete-lhss C D L ⟹ *strong-quasi-reducible C D L*
⟨*proof*⟩

lemma *arg-imp-subt*: $s \in set\ (args\ t) \Longrightarrow t \trianglerighteq s$
⟨*proof*⟩

lemma *strong-quasi-reducible-imp-quasi-reducible*:
strong-quasi-reducible C D L ⟹ *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** $L :: ('f,'v)term\ set$
assumes $\bigwedge l\ f\ ls\ \tau s\ \tau.\ l \in L \Longrightarrow l = Fun\ f\ ls \Longrightarrow \neg\ f : \tau s \to \tau\ in\ C$
shows *pat-complete-lhss C D L* ⟷ *quasi-reducible C D L*
⟨*proof*⟩

end

10 Setup for Experiments

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

turn error message into runtime error

definition *pat-complete-alg* :: $(('f \times 's\ list) \times 's)list \Rightarrow (('f \times 's\ list) \times 's)list \Rightarrow ('f,'v)term\ list \Rightarrow bool$ **where**

pat-complete-alg C D lhss = (
case decide-pat-complete-lhss C D lhss of Inl err ⇒ *Code.abort* (*err* (*STR* ''''))
(λ -. *True*)
| *Inr res* ⇒ *res*)

turn error message into runtime error

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

Examples

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

definition *rn-string* **where** *rn-string x* = ''*x*'' @ *show* (*x* :: *nat*)
definition *rv-string* **where** *rv-string x* = ''*y*'' @ *x*

lemma *renaming-string*: *renaming-funs rn-string rv-string*
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *decide-pat-complete-wrapper*:
assumes (*case decide-pat-complete-lhss C D lhss of Inr b* ⇒ *Some b* | *Inl -* ⇒ *None*) = *Some res*
shows *pat-complete-lhss* (*map-of C*) (*map-of D*) (*set lhss*) = *res*
⟨*proof*⟩

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

lemma *decide-strong-quasi-reducible-wrapper*:
assumes (*case decide-strong-quasi-reducible C D lhss of Inr b* ⇒ *Some b* | *Inl -* ⇒ *None*) = *Some res*
shows *strong-quasi-reducible* (*map-of C*) (*map-of D*) (*set lhss*) = *res*
⟨*proof*⟩

lemma *pat-complete-lhss* (*map-of nat-bool*) (*map-of even-nat*) (*set even-lhss*)
⟨*proof*⟩

lemma ¬ *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 ¬ *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

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

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

lemma ¬ *pat-complete-lhss* (*map-of nat-bool*) (*map-of* [((''*f*'',[''*nat*'',''*nat*'',''*nat*'']),''*bool*'')])
(*set non-lin-lhss*)
  ⟨*proof*⟩

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

definition *testproblem* (*c* :: *nat*) *n* = (*let s* = *String.implode*; *s* = *id*;
    c1 = *even c*;
    c2 = *even* (*c div 2*);
    c3 = *even* (*c div 4*);
    c4 = *even* (*c div 8*);
    revo = (*if c4 then id else rev*);
    nn = [*0* ..< *n*];
    rnn = (*if c4 then id nn else rev nn*);
    b = *s* ''*b*''; *t* = *s* ''*tt*''; *f* = *s* ''*ff*''; *g* = *s* ''*g*'';
    gg = (λ *ts*. *Fun g* (*revo ts*));
    ff = *Fun f* [];
    tt = *Fun t* [];
    C = [((*t*, [] :: *string list*), *b*), ((*f*, []), *b*)];

D = [((g, replicate (2 ∗ n) b), b)];
x = (λ i :: nat. Var (s (''x'' @ show i)));
y = (λ i :: nat. Var (s (''y'' @ show i)));
lhsF = gg (if c1 then List.maps (λ i. [ff, y i]) rnn else (replicate n ff @ map y rnn));
lhsT = (λ b j. gg (if c1 then List.maps (λ i. if i = j then [tt, b] else [x i, y i]) rnn else
(map (λ i. if i = j then tt else x i) rnn @ map (λ i. if i = j then b else y i) rnn)));
lhssT = (if c2 then List.maps (λ i. [lhsT tt i, lhsT ff i]) nn else List.maps (λ b. map (lhsT b) nn) [tt,ff]);
lhss = (if c3 then [lhsF] @ lhssT else lhssT @ [lhsF])
in (C, D, lhss))

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

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

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

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

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

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

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

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

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

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

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

tree automata encoding

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

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

we further fix the parameters to generate the example TRSs

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

definition *tt* = *cSym* (*STR* ''tt'') *0*
definition *ff* = *cSym* (*STR* ''ff'') *0*

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

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

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

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

definition automataLeft = cAut signature leftStates finStates (cRule g (replicate
(2 * nat-of-integer n) qb) qfin # qbRules)
definition automataRight = (case test-problem-integer c n perms of
 (-,-,lhss) ⇒ cAut signature rightStates finStates (map termToRule lhss @ stdRules))

definition encodeAutomata = (automataLeft, automataRight)

definition patCompleteAutomataTest = (checkSubset automataLeft automataRight)

end
end

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

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

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

definition showAutomata where showAutomata n c perms = (case encodeAu-
tomata id (λ n a. n)
  (λ f qs q. paren f (f +++ STR ''('') (STR '')'') (STR '','') qs +++ STR '' ->
```

'' +++ q)
(λ sig Q Qfin rls.
STR ''tree−automata has final states: '' +++ paren (STR ''{}'') (STR ''{'') (STR ''}'') (STR '','') Qfin +++ STR ''↩''
+++ STR ''and transitions:↩'' +++ paren (STR '''') (STR '''') (STR '''') (STR ''↩'') rls +++ STR ''↩↩'') n c perms
of (all,pats) ⇒ STR ''decide whether language of first automaton is subset of the second automaton↩↩''
+++ STR ''first '' +++ all +++ STR ''↩and second '' +++ pats)

value *showAutomata 4 4* []

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

value *createHaskellInput 4 4* []

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

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

end

References

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

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

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

[4] R. Thiemann and A. Yamada. A verified algorithm for deciding pattern completeness. In J. Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13,*

2024, Tallinn, Estonia, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. To appear.