Verifying a Decision Procedure for Pattern Completeness*

René Thiemann

University of Innsbruck, Austria

Akihisa Yamada

National Institute of Advanced Industrial Science and Technology, Japan

June 3, 2024

Abstract

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

Contents

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

1 Introduction

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

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

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

2 Pattern Completeness

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

3 A Set-Based Inference System to Decide Pattern Completeness

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

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

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

3.1 Definition of Algorithm – Inference Rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The transformation rules of the paper.

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

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

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

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

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

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

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

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

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

3.2 Soundness of the inference rules

The empty set of variables

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

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

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

A definition of pattern completeness for pattern problems.

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

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

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

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

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

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

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

lemma *type-conversion*: $t : s\ in\ \mathcal{T}(C,\emptyset) \Longrightarrow t \cdot \sigma : s\ in\ \mathcal{T}(C,\emptyset)$
⟨*proof*⟩

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* |- *p*) $\leq$ *size t*
⟨*proof*⟩

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*: **fixes** *x* :: *'v*
shows *Conflict-Clash s t* $\Longrightarrow$ $\exists$ *p*. *p* $\in$ *poss s* $\wedge$ *p* $\in$ *poss t* $\wedge$ *is-Fun* (*s* |-*p*) $\wedge$ *is-Fun* (*t* |-*p*) $\wedge$ *root* (*s* |-*p*) $\neq$ *root* (*t* |- *p*) (**is** *?B1* $\Longrightarrow$ *?B2*)
and *Conflict-Var s t x* $\Longrightarrow$
$\exists$ *p* . *p* $\in$ *poss s* $\wedge$ *p* $\in$ *poss t* $\wedge$ *s* |-*p* $\neq$ *t* |-*p* $\wedge$ (*s* |-*p* = *Var x* $\vee$ *t* |-*p* = *Var x*) (**is** *?C1 x* $\Longrightarrow$ *?C2 x*)
and *s* $\neq$ *t* $\Longrightarrow$ $\exists$ *x*. *Conflict-Clash s t* $\vee$ *Conflict-Var s t x*
and *Conflict-Var s t x* $\Longrightarrow$ *x* $\in$ *vars s* $\cup$ *vars t*
and *conflicts s t* = *Some* [] $\longleftrightarrow$ *s* = *t* (**is** *?A*)
⟨*proof*⟩

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

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

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

locale *pattern-completeness-context-with-assms* = *pattern-completeness-context S C m Cl inf-sort ty*
for *S* **and** *C* :: (*'f*,*'s*)*ssig*
and *m Cl inf-sort*
and *ty* :: *'v itself* +
assumes *sorts-non-empty*: $\bigwedge$ *s*. *s* $\in$ *S* $\Longrightarrow$ $\exists$ *t*. *t* : *s in* $\mathcal{T}$(*C*, *EMPTY*)
and *C-sub-S*: $\bigwedge$ *f ss s*. *f* : *ss* $\rightarrow$ *s in C* $\Longrightarrow$ *insert s* (*set ss*) $\subseteq$ *S*
and *m*: $\bigwedge$ *f ss s*. *f* : *ss* $\rightarrow$ *s in C* $\Longrightarrow$ *length ss* $\leq$ *m*
and *inf-sort-def*: *s* $\in$ *S* $\Longrightarrow$ *inf-sort s* = ($\neg$ *bdd-above* (*size* ‘ {*t* . *t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*)}))
and *Cl*: $\bigwedge$ *s*. *set* (*Cl s*) = {(*f*,*ss*). *f* : *ss* $\rightarrow$ *s in C*}
and *Cl-len*: $\bigwedge$ σ. *Ball* (*length* ‘ *snd* ‘ *set* (*Cl* σ)) (λ *a*. *a* $\leq$ *m*)

begin

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

Preservation of well-formedness

lemma *mp-step-wf*: $mp \rightarrow_s mp' \Longrightarrow$ *wf-match* $mp \Longrightarrow$ *wf-match* mp'
⟨*proof*⟩

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

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

Soundness requires some preparations

lemma *cg-exists*: $\exists\ \sigma g.$ *cg-subst* σg
⟨*proof*⟩

definition σg :: $('f,nat \times 's,'v)gsubst$ **where** $\sigma g = (SOME\ \sigma.\ cg\text{-}subst\ \sigma)$

lemma σg: *cg-subst* σg ⟨*proof*⟩

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

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

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

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

⟨*proof*⟩

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

⟨*proof*⟩

lemma *term-of-sort*: **assumes** $s \in S$
shows $\exists\ t.\ t : s\ in\ \mathcal{T}(C,EMPTYn)$
⟨*proof*⟩

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

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

lemma *mp-fail-pcorrect*: *mp-fail mp* $\Longrightarrow \neg$ *match-complete-wrt* σ *mp*
⟨*proof*⟩

lemma *pp-step-pcorrect*: $pp \Rightarrow_s pp' \Longrightarrow$ *pat-complete pp* = *pat-complete* pp'
⟨*proof*⟩

lemma *pp-success-pcorrect*: *pp-success pp* $\Longrightarrow$ *pat-complete pp*
⟨*proof*⟩

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

4 A Multiset-Based Inference System to Decide Pattern Completeness

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

4.1 Definition of the Inference Rules

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

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

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

abbreviation *mp-mset* :: $('f,'v,'s)$*match-problem-mset* ⇒ $('f,'v,'s)$*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* ⇒ $('f,'v,'s)$*match-problem-mset* ⇒ $('f,'v,'s)$*match-problem-mset* **where**
subst-match-problem-mset τ = *image-mset* (*subst-left* τ)

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

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

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

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

inductive *pp-step-mset* :: $('f,'v,'s)$*pat-problem-mset* ⇒ $('f,'v,'s)$*pats-problem-mset* ⇒ *bool*
(**infix** $\Rightarrow_m$ *50*) **where**
pat-remove-pp: *add-mset* {#} *pp* $\Rightarrow_m$ {#}

| *pat-simp-mp*: *mp-step-mset mp mp′* ⟹ *add-mset mp pp* $\Rightarrow_m$ {# (*add-mset mp′ pp*) #}
| *pat-remove-mp*: *match-fail mp* ⟹ *add-mset mp pp* $\Rightarrow_m$ {# *pp* #}
| *pat-instantiate*: *tvars-disj-pp* {*n* ..< *n*+*m*} (*pat-mset* (*add-mset mp pp*)) ⟹
(*Var x*, *l*) ∈ *mp-mset mp* ∧ *is-Fun l* ∨
(*s*, *Var y*) ∈ *mp-mset mp* ∧ (*t*, *Var y*) ∈ *mp-mset mp* ∧ *Conflict-Var s t x* ∧ ¬ *inf-sort* (*snd x*) ⟹
add-mset mp pp $\Rightarrow_m$ *mset* (*map* (λ τ. *subst-pat-problem-mset* τ (*add-mset mp pp*)) (τ*s-list n x*))

inductive *pat-fail* :: (′*f*,′*v*,′*s*)*pat-problem-mset* ⇒ *bool* **where**
pat-failure′: *Ball* (*pat-mset pp*) *inf-var-conflict* ⟹ *pat-fail pp*
| *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′*}

4.2 The evaluation cannot get stuck

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

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

lemma *pat-fail-or-trans*:
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 *NF*: *P* ∈ *NF* ⇛
shows *P* ∈ {{#}, *bottom-mset*}
⟨*proof*⟩
end

4.3 Termination

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

fun *fun-diff* :: $('f,'v)term \Rightarrow ('f,'w)term \Rightarrow nat$ **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*: $(\bigwedge y.\ y \in\# N \Longrightarrow (y,x) \in R) \Longrightarrow (N + M,\ add\text{-}mset\ x\ M) \in mult\ R$
⟨*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* :: $'a \Rightarrow nat$) (*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* :: $('a \Rightarrow nat) \Rightarrow 'a\ multiset \Rightarrow nat$ **where**
sum-ms f ms ≡ *sum-mset* (*image-mset f ms*)

definition *meas-diff* :: $('f,'v,'s)pat\text{-}problem\text{-}mset \Rightarrow nat$ **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* 𝒯(*C*,*EMPTYn*)}) *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-mp* (*mp-mset mp*)))

definition *meas-symbols* :: (′*f*,′*v*,′*s*)*pat-problem-mset* ⇒ *nat* **where**
meas-symbols = *sum-ms size-mset*

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

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

lemma *tvars-mp-mono*: *mp* ⊆# *mp*′ ⟹ *tvars-mp* (*mp-mset mp*) ⊆ *tvars-mp* (*mp-mset mp*′)
⟨*proof*⟩

lemma *meas-finvars-mono*: **assumes** *tvars-mp* (*mp-mset mp*) ⊆ *tvars-mp* (*mp-mset*

mp')
shows *meas-finvars* {#*mp*#} ≤ *meas-finvars* {#*mp'*#}
⟨*proof*⟩

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

lemma *rel-mp-mp-step-mset*:
assumes $mp \rightarrow_m mp'$
shows {#*mp*#} ≻ {#*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*) ≤ *meas-diff p*
⟨*proof*⟩

lemma *meas-sub*: **assumes** *sub*: *p'* ⊆# *p*
shows *meas-diff p'* ≤ *meas-diff p*
meas-finvars p' ≤ *meas-finvars p*
meas-symbols p' ≤ *meas-symbols p*
⟨*proof*⟩

lemma *meas-sub-rel-pat*: **assumes** *sub*: *p'* ⊂# *p*
shows *p* ≻ *p'*
⟨*proof*⟩

lemma *max-size-term-of-sort*: **assumes** *sS*: *s* ∈ *S* **and** *inf*: ¬ *inf-sort s*
shows ∃ *t*. *t* : *s in* $\mathcal{T}$(*C*,*EMPTYn*) ∧ *max-size s* = *size t* ∧ (∀ *t'*. *t'* : *s in* $\mathcal{T}$(*C*,*EMPTYn*) ⟶ *size t'* ≤ *size t*)
⟨*proof*⟩

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

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

lemma *rel-pp-step-mset*:
assumes $p \Rightarrow_m ps$
and *p'* ∈# *ps*
shows *p* ≻ *p'*
⟨*proof*⟩

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

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

termination of the multiset based implementation

theorem *SN-P-step*: $SN \Rrightarrow$
$\langle proof \rangle$

4.4 Partial Correctness via Refinement

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

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

lemma *mp-step-mset-mp-trans*: $mp \rightarrow_m mp' \Longrightarrow mp\text{-}mset\ mp \rightarrow_s mp\text{-}mset\ mp'$
$\langle proof \rangle$

lemma *mp-fail-cong*: $mp\text{-}fail\ mp \Longrightarrow mp = mp' \Longrightarrow mp\text{-}fail\ mp'$ $\langle proof \rangle$

lemma *match-fail-mp-fail*: $match\text{-}fail\ mp \Longrightarrow mp\text{-}fail\ (mp\text{-}mset\ mp)$
$\langle proof \rangle$

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

lemma *P-step-mset-imp-set*: **assumes** $P \Rrightarrow_m Q$
shows $pats\text{-}mset\ P \Rrightarrow_s pats\text{-}mset\ Q$
$\langle proof \rangle$

lemma *P-step-pp-trans*: **assumes** $(P,Q) \in \Rrightarrow$
shows $pats\text{-}mset\ P \Rrightarrow_s pats\text{-}mset\ Q$
$\langle proof \rangle$

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

corollary *P-steps-pcorrect*: **assumes** *wf*: *wf-pats* (*pats-mset* P)
and *step*: $(P,Q) \in \Rrightarrow^*$
shows *wf-pats* (*pats-mset* Q) $\wedge$ (*pats-complete* (*pats-mset* P) $\longleftrightarrow$ *pats-complete* (*pats-mset* Q))
$\langle proof \rangle$

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

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

end
end

5 Computing Nonempty and Infinite sorts

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

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

5.1 Deciding the nonemptyness of all sorts under consideration

function *compute-nonempty-main* :: $'\tau$ *set* ⇒ (($'f$ × $'\tau$ *list*) × $'\tau$) *list* ⇒ $'\tau$ *set*
where
compute-nonempty-main ne ls = (*let rem-ls* = *filter* (λ *f*. *snd f* ∉ *ne*) *ls in*
case partition (λ ((-,*args*),-). *set args* ⊆ *ne*) *rem-ls of*
(*new*, *rem*) ⇒ *if new* = [] *then ne else compute-nonempty-main* (*ne* ∪ *set* (*map snd new*)) *rem*)
⟨*proof*⟩

termination
⟨*proof*⟩

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

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

lemma *compute-nonempty-sorts*:
assumes *distinct* (*map fst Cs*)
and *map-of Cs* = *C*

shows *compute-nonempty-sorts Cs* = $\{\tau.\ \exists\ t :: ('f,'v)term.\ t : \tau\ in\ \mathcal{T}(C,\emptyset)\}$ (**is** - = *?NE*)
⟨*proof*⟩

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

lemma *decide-nonempty-sorts*:
assumes *distinct* (*map fst Cs*)
and *map-of Cs* = *C*
shows *decide-nonempty-sorts* τs *Cs* = *None* $\Longrightarrow \forall\ \tau \in set\ \tau s.\ \exists\ t :: ('f,'v)term.\ t : \tau\ in\ \mathcal{T}(C,\emptyset)$
decide-nonempty-sorts τs *Cs* = *Some* $\tau \Longrightarrow \tau \in set\ \tau s \wedge \neg\ (\exists\ t :: ('f,'v)term.\ t : \tau\ in\ \mathcal{T}(C,\emptyset))$
⟨*proof*⟩

5.2 Deciding infiniteness of a sort

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

function *compute-inf-main* :: $'\tau\ set \Rightarrow ('\tau \times ('f \times '\tau\ list)list)\ list \Rightarrow '\tau\ set$ **where**
compute-inf-main m-inf ls = (
let (*fin*, *ls'*) =
partition $(\lambda\ (\tau,fs).\ \forall\ \tau s \in set\ (map\ snd\ fs).\ \forall\ \tau \in set\ \tau s.\ \tau \notin m\text{-}inf)\ ls$
in if fin = [] *then m-inf else compute-inf-main* (*m-inf* − *set* (*map fst fin*)) *ls'*)
⟨*proof*⟩

termination
⟨*proof*⟩

lemma *compute-inf-main*: **fixes** $E :: 'v \rightharpoonup 't$ **and** $C :: ('f,'t)ssig$
assumes *E*: $E = \emptyset$
and *C-Cs*: *C* = *map-of Cs'*
and *Cs'*: *set Cs'* = *set* (*concat* (*map* $((\lambda\ (\tau, fs).\ map\ (\lambda\ f.\ (f,\tau))\ fs))$ *Cs*))
and *arg-types-inhabitet*: $\forall\ f\ \tau s\ \tau\ \tau'.\ f : \tau s \to \tau\ in\ C \longrightarrow \tau' \in set\ \tau s \longrightarrow (\exists\ t.\ t : \tau'\ in\ \mathcal{T}(C,E))$
and *dist*: *distinct* (*map fst Cs*) *distinct* (*map fst Cs'*)
and *inhabitet*: $\forall\ \tau\ fs.\ (\tau,fs) \in set\ Cs \longrightarrow set\ fs \neq \{\}$
and $\forall\ \tau.\ \tau \notin m\text{-}inf \longrightarrow bdd\text{-}above\ (size\ `\ \{t.\ t : \tau\ in\ \mathcal{T}(C,E)\})$
and *set ls* ⊆ *set Cs*
and *fst* ‘ (*set Cs* − *set ls*) ∩ *m-inf* = {}
and *m-inf* ⊆ *fst* ‘ *set ls*
shows *compute-inf-main m-inf ls* = $\{\tau.\ \neg\ bdd\text{-}above\ (size\ `\ \{t.\ t : \tau\ in\ \mathcal{T}(C,E)\})\}$

⟨*proof*⟩

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

lemma *compute-inf-sorts*:
fixes $E :: 'v \rightharpoonup 't$ **and** $C :: ('f,'t)ssig$
assumes *E*: $E = \emptyset$
and *C-Cs*: *C* = *map-of Cs*
and *arg-types-inhabitet*: $\forall\ f\ \tau s\ \tau\ \tau'.\ f : \tau s \rightarrow \tau\ in\ C \longrightarrow \tau' \in set\ \tau s \longrightarrow (\exists\ t.\ t : \tau'\ in\ \mathcal{T}(C,E))$
and *dist*: *distinct* (*map fst Cs*)
shows *compute-inf-sorts Cs* = $\{\tau.\ \neg\ bdd\text{-}above\ (size\ `\ \{t.\ t : \tau\ in\ \mathcal{T}(C,E)\})\}$
⟨*proof*⟩
end

6 A List-Based Implementation to Decide Pattern Completeness

theory *Pattern-Completeness-List*
imports
Pattern-Completeness-Multiset
Compute-Nonempty-Infinite-Sorts
HOL−Library.AList
begin

6.1 Definition of Algorithm

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

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

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

where *mp-list* ≡ *mset*

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

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

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

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

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

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

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

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

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

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

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

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

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

definition *find-var* :: *('f,'v,'s)match-problem-lr list* ⇒ - **where** *find-var p = (case concat (map (λ (lx,-). lx) p) of*

(*x*,*t*) # - ⇒ *x*
| [] ⇒ *let* (-,*rx*,*b*) = *hd* (*filter* (*Not o snd o snd*) *p*)
in case hd rx of (*x*, *s* # *t* # -) ⇒ *hd* (*the* (*conflicts s t*)))

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

context *pattern-completeness-context*
begin

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

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

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

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

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

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

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

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

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

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

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

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

6.2 Partial Correctness of the Implementation

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

lemma *mset-concat-union*:

mset (*concat xs*) = $\sum_{\#}$ (*mset* (*map mset xs*))
⟨*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*⟩

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-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-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-pat-lr* :: (′*f*,′*v*,′*s*)*pat-problem-lr* ⇒ *bool* **where**
wf-pat-lr mps = (*Ball* (*set mps*) (λ *mp*. *wf-lr2 mp* ∧ ¬ *empty-lr mp*))

lemma *mp-step-mset-cong*:
assumes $(\rightarrow_m)^{**}$ *mp mp'*
shows (*add-mset* (*add-mset mp p*) *P*, *add-mset* (*add-mset mp' p*) *P*) $\in \Rrightarrow^*$
⟨*proof*⟩

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

lemma *mp-step-mset-steps-vars*: **assumes** $(\rightarrow_m)^{**}$ *mp mp'*
shows *tvars-mp* (*mp-mset mp*) $\supseteq$ *tvars-mp* (*mp-mset mp'*)
⟨*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'* $\Longrightarrow$ $(\rightarrow_m)^{**}$ (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*)) (*mp-lr* (*ls*,*rx'*) + *M*) $\wedge$ *wf-rx rx'*
res = *None* $\Longrightarrow$ *match-fail* (*add-mset* (*t*, *Var x*) (*mp-lr mp* + *M*))
⟨*proof*⟩

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

lemma *match-var-impl*: **assumes** *wf*: *wf-lr mp*
shows $(\rightarrow_m)^{**}$ (*mp-lr mp*) (*mp-lr* (*match-var-impl mp*))
and *wf-lr2* (*match-var-impl mp*)
⟨*proof*⟩

lemma *match-steps-impl*: **assumes** *match-steps-impl mp* = *res*
shows *res* = *Some mp'* $\Longrightarrow$ $(\rightarrow_m)^{**}$ (*mp-list mp*) (*mp-lr mp'*) $\wedge$ *wf-lr2 mp'*
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 p pd* = *res*
and *wf-pat-lr pd*
and *tvars-pp* (*pat-mset* (*pat-mset-list p* + *pat-lr pd*)) $\subseteq$ *V*
shows *res* = *None* $\Longrightarrow$ (*add-mset* (*pat-mset-list p* + *pat-lr pd*) *P*, *P*) $\in \Rrightarrow^+$
and *res* = *Some p'* $\Longrightarrow$ (*add-mset* (*pat-mset-list p* + *pat-lr pd*) *P*, *add-mset* (*pat-lr p'*) *P*) $\in \Rrightarrow^*$
$\wedge$ *wf-pat-lr p'* $\wedge$ *tvars-pp* (*pat-mset* (*pat-lr p'*)) $\subseteq$ *V*
⟨*proof*⟩

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

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

lemma *pat-impl*: **assumes** *pat-impl n p = res*
  and *vars*: *fst ‘ tvars-pp (pat-list p)* $\subseteq$ {..<*n*}
 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*) $\in$ $\Rrightarrow^+$
   $\wedge$ *fst ‘ tvars-pp* ($\bigcup$ (*pat-list ‘ set ps*)) $\subseteq$ {..<*n*+*m*}
⟨*proof*⟩

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

lemma *pats-impl-P-step*: **assumes** *Ball* (*set ps*) (λ *p*. *fst ‘ tvars-pp* (*pat-list p*) $\subseteq$ {..<*n*})
 shows
  — if result is True, then one can reach empty set
  pats-impl n ps $\Longrightarrow$ (*pats-mset-list ps*, {#}) $\in$ $\Rrightarrow^*$
  — if result is False, then one can reach bottom
  $\neg$ *pats-impl n ps* $\Longrightarrow$ (*pats-mset-list ps*, *bottom-mset*) $\in$ $\Rrightarrow^*$
⟨*proof*⟩

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

theorem *pats-impl*: **assumes** *wf*: $\forall$ *pp* $\in$ *pat-list ‘ set P*. *wf-pat pp*
 and *n*: $\forall$ *p*$\in$*set P*. *fst ‘ tvars-pp* (*pat-list p*) $\subseteq$ {..<*n*}
 shows *pats-impl n P* $\longleftrightarrow$ *pats-complete* (*pat-list ‘ set P*)
⟨*proof*⟩

corollary *pat-complete-impl*:
 assumes *wf*: *snd ‘* $\bigcup$ (*vars ‘ fst ‘ set* (*concat* (*concat P*))) $\subseteq$ *S*
 shows *pat-complete-impl P* $\longleftrightarrow$ *pats-complete* (*pat-list ‘ set P*)
⟨*proof*⟩
end

6.3 Getting the result outside the locale with assumptions

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

context *pattern-completeness-context*
begin
lemma *pat-complete-impl-wrapper*: **assumes** *C-Cs*: *C = map-of Cs*
 and *dist*: *distinct* (*map fst Cs*)
 and *inhabited*: *decide-nonempty-sorts Sl Cs = None*
 and *S-Sl*: *S = set Sl*
 and *inf-sort*: *inf-sort* = (λ *s*. *s* $\in$ *compute-inf-sorts Cs*)
 and *C*: $\bigwedge$ *f* σs σ. ((*f*,σs),σ) $\in$ *set Cs* $\Longrightarrow$ *length* $\sigma s \leq m$ $\wedge$ *set* (σ # σs) $\subseteq$ *S*
 and *Cl*: $\bigwedge$ *s*. *Cl s = map fst* (*filter* ((=) *s o snd*) *Cs*)
 and *P*: *snd ‘* $\bigcup$ (*vars ‘ fst ‘ set* (*concat* (*concat P*))) $\subseteq$ *S*

shows *pat-complete-impl P* = *pats-complete* (*pat-list* ‘ *set P*)
⟨*proof*⟩
end

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

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

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

definition *decide-pat-complete* :: (('*f* × '*s list*) × '*s*)*list* ⇒ ('*f*,'*v*,'*s*)*pats-problem-list* ⇒ *bool* **where**
decide-pat-complete Cs P = (*let Sl* = *sorts-of-ssig-list Cs*;
m = *max-list* (*map* (*length o snd o fst*) *Cs*);
Cl = (λ *s*. *map fst* (*filter* ((=) *s* ∘ *snd*) *Cs*));
IS = *compute-inf-sorts Cs*
in pattern-completeness-context.pat-complete-impl m Cl (λ *s*. *s* ∈ *IS*)) *P*

abbreviation (*input*) *pat-complete* **where**
pat-complete ≡ *pattern-completeness-context.pat-complete*

abbreviation (*input*) *pats-complete* **where**
pats-complete ≡ *pattern-completeness-context.pats-complete*

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

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

end

7 Pattern-Completeness and Related Properties

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

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

begin

A pattern completeness decision procedure for a set of lhss

definition *basic-terms* :: ('f,'s)ssig ⇒ ('f,'s)ssig ⇒ ('v ⇀ 's) ⇒ ('f,'v)term set (𝓑'(-,-,-')) **where**
  𝓑(C,D,V) = { Fun f ts | f ss s ts . f : ss → s in D ∧ ts :$_l$ ss in 𝒯(C,V)}

definition *matches* :: ('f,'v)term ⇒ ('f,'v)term ⇒ bool (**infix** matches 50) **where**
  l matches t = (∃ σ. t = l · σ)

definition *pat-complete-lhss* :: ('f,'s)ssig ⇒ ('f,'s)ssig ⇒ ('f,'v)term set ⇒ bool **where**
  pat-complete-lhss C D L = (∀ t ∈ 𝓑(C,D,∅). ∃ l ∈ L. l matches t)

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 (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'')));
    let pats = [Fun f (map Var (zip [0..<length ss] ss)). ((f,ss),s) ← D];
    let P = [[[(pat,lhs)]. lhs ← lhss]. pat ← pats];
    return (decide-pat-complete C P)
  }

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

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 ∈ 𝓑(C,D,∅). ∃ ti ∈ set (t # args t). ∃ l ∈ L. l matches ti)

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

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

lemma *decide-strong-quasi-reducible*:
  assumes *decide-strong-quasi-reducible C D* (*lhss* :: $('f,'v)term\ list$) = *return b*
  shows *b = strong-quasi-reducible* (*map-of C*) (*map-of D*) (*set lhss*)
⟨*proof*⟩

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

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

lemma *pat-complete-imp-strong-quasi-reducible*:
  pat-complete-lhss C D L $\Longrightarrow$ *strong-quasi-reducible C D L*
  ⟨*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 $\Longrightarrow$ *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* $\longleftrightarrow$ *quasi-reducible C D L*

⟨*proof*⟩

end

8 Setup for Experiments

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

turn error message into runtime error

definition *pat-complete-alg* :: (('f × 's list) × 's)list ⇒ (('f × 's list) × 's)list ⇒ ('f,'v)term list ⇒ 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 *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-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*⟩

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*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

tree automata encoding

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

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

we further fix the parameters to generate the example TRSs

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

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

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

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

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

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

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

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

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

end
end

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

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

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

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

value *showAutomata 4 4* []

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

value *createHaskellInput 4 4* []

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

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

end

References

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

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

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

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