A Verified Algorithm for Deciding Pattern Completeness

René Thiemann
University of Innsbruck, Austria

Akihisa Yamada
National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

Abstract

Pattern completeness is the property that the left-hand sides of a functional program cover all cases w.r.t. pattern matching. In the context of term rewriting a related notion is quasi-reducibility, a prerequisite if one wants to perform ground confluence proofs by rewriting induction.

In order to certify such confluence proofs, we develop a novel algorithm that decides pattern completeness and that can be used to ensure quasi-reducibility. One of the advantages of the proposed algorithm is its simple structure: it is similar to that of a regular matching algorithm and, unlike an existing decision procedure for quasi-reducibility, it avoids enumerating all terms up to a given depth.

Despite the simple structure, proving the correctness of the algorithm is not immediate. Therefore we formalize the algorithm and verify its correctness using the proof assistant Isabelle/HOL. To this end, we not only verify some auxiliary algorithms, but also design an Isabelle library on sorted term rewriting. Moreover, we export the verified code in Haskell and experimentally evaluate its performance. We observe that our algorithm significantly outperforms existing algorithms, even including the pattern completeness check of the GHC Haskell compiler.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Program verification; Theory of computation → Higher order logic

Keywords and phrases Isabelle/HOL, pattern matching, term rewriting

Digital Object Identifier 10.4230/LIPIcs.FSCD.2024.27

Supplementary Material *InteractiveResource (Website with Formalization, Experiments, Sources)*: `http://cl-informatik.uibk.ac.at/software/ceta/experiments/pat_complete/`

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

Acknowledgements We thank Takahito Aoto and Fabian Mitterwallner for their help in conducting experiments with the tools AGCP and FORT-h, respectively; and we thank Dohan Kim for his contributions to the formalization of an auxiliary algorithm. We are grateful to all reviewers for their helpful remarks and suggestions.

1 Introduction

Consider programs written in a declarative style such as functional programs or term rewrite systems (TRSs), where evaluation is triggered by pattern matching. In many applications it is important to ensure that evaluation of a given program cannot get stuck – this property is called quasi-reducibility [8] in the context of TRSs or pattern completeness in the context of functional programming. For instance in Isabelle/HOL [14], in a function definition the patterns must cover all cases (in addition to termination), since HOL is a logic of total functions. Moreover, automated theorem proving methods that are based on rewriting induction [1, 15] require similar completeness results, e.g., for proving ground confluence.

▶ **Example 1.** Let $\mathcal{C}_{\mathbb{N}} = \{\mathsf{true} : \mathbb{B}, \mathsf{false} : \mathbb{B}, \mathsf{0} : \mathbb{N}, \mathsf{s} : \mathbb{N} \to \mathbb{N}\}$ be the set of constructors to represent the Booleans and natural numbers in Peano's notation. We consider a TRS $\mathcal{R}_{\mathbb{N}}$ that defines a function $\mathsf{even} : \mathbb{N} \to \mathbb{B}$ to compute whether a natural number is even.

$$\mathsf{even}(\mathsf{0}) \to \mathsf{true} \qquad \mathsf{even}(\mathsf{s}(\mathsf{0})) \to \mathsf{false} \qquad \mathsf{even}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{even}(x) \tag{1}$$

This TRS is quasi-reducible, since no matter which number n we provide as argument, one of the left-hand sides (lhss) will match the term $\mathsf{even}(n)$; this fact can easily be seen by a case-analysis on whether n represents 0, 1, or some larger number. Note the importance of sorts:[1] without them, the evaluation of the (unsorted) term $\mathsf{even}(\mathsf{s}(\mathsf{true}))$ would get stuck.

Kapur et al. proved the decidability of quasi-reducibility [8]. Their decidability result does not yield a practical algorithm: it has an exponential best-case complexity, i.e., to ensure quasi-reducibility, one always has to enumerate exponentially many terms and test whether their evaluation does not get stuck. Therefore, Lazrek, Lescanne and Thiel developed a more practical approach. Their *complement algorithm* [12] is a decision procedure for pattern completeness in the left-linear case, but it might fail on TRSs that are not left-linear. Note that in the left-linear case, pattern completeness and quasi-reducibility can also be encoded into a problem about tree automata.

In this paper, we develop a novel algorithm for pattern completeness with the following key features.

- It is a decision procedure, even in the non-linear case.
- The algorithm is syntax directed and it is easy to implement.
- In our experiments it outperforms existing implementations of the complement algorithm, the approach via tree automata, and pattern completeness check by the `ghc` Haskell compiler.
- Its correctness is fully verified in Isabelle/HOL.

We are aware of two other algorithms to ensure quasi-reducibility in more complex settings, e.g., where rules may be guarded by arithmetic constraints such as "this rule is only applicable if $x > 0$" [5, 9], but both algorithms do not properly generalize the result of Kapur et al. since they are restricted to linear lhss. Bouhoula and Jacquemard [3] also designed an algorithm in a more complex setting with conditions and constraints, and a back-end that is based on constrained tree automata techniques. Since their soundness result is restricted to ground confluent systems, their algorithm is not applicable in our use case; ultimately we want to verify ground confluence proofs on methods that rely upon quasi-reducibility. Moreover, Bouhoula developed an algorithm to verify ground confluence and completeness at the same time [2], where we are not sure whether it can also be adjusted to an algorithm that just ensures completeness, e.g., for non-ground confluent systems. Furthermore, there are proof methods that ensure pattern completeness within proof assistants. These are used to ensure well-definedness of function definitions. For instance, in Isabelle/HOL there is a corresponding method `pat_completeness` [10], but as many other algorithms for pattern completeness, it is restricted to the left-linear case.

There are also algorithms to compile pattern matching [11, 16], however these have a different focus: their major aim is not to decide or to ensure completeness, but instead they generate efficient code for functional programs that are defined by pattern matching.

[1] A *sort* in the TRS context is the same as a *type* when speaking about functional programs. Since most of this paper is written using TRS notation, we speak of sorts instead of types in the rest of the paper.

The paper is organized as follows: In Section 2 on preliminaries we introduce notions and notations, and recall the core concepts of pattern completeness and quasi-reducibility. Then in Section 3 we present the first part of our novel algorithm that covers the linear case. The algorithm is then extended to handle the general case in Section 4. Afterwards we present details on the Isabelle formalization and on the implementation in Section 5. The experimental results are provided in Section 6 before we conclude in Section 7.

The formalization, the executable code and details on the experiments are available at:

`http://cl-informatik.uibk.ac.at/software/ceta/experiments/pat_complete/`

2 Preliminaries

We fix a set $\mathcal{S}$ of sorts. A sorted set A is a set where each element a is associated with a sort $\iota \in \mathcal{S}$, written $a : \iota \in A$. A sorted signature $\mathcal{F}$ is a set of function symbols f, each associated with a nonempty sequence of sorts $\iota_1, \dots, \iota_n, \iota_0 \in \mathcal{S}$, written $f : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{F}$. Given a sorted signature $\mathcal{F}$ and a sorted set $\mathcal{V}$ of variables, the sorted set $\mathcal{T}(\mathcal{F},\mathcal{V})$ of terms is defined as follows: $x : \iota \in \mathcal{T}(\mathcal{F},\mathcal{V})$ if $x : \iota \in \mathcal{V}$; and $f(t_1,\dots,t_n) : \iota_0 \in \mathcal{T}(\mathcal{F},\mathcal{V})$ if $f : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{F}$ and $t_1 : \iota_1, \dots, t_n : \iota_n \in \mathcal{T}(\mathcal{F},\mathcal{V})$. We denote the set of variables occurring in t by $\mathsf{Var}(t)$. By $\mathcal{T}(\mathcal{F})$ we denote the sorted set of ground terms, i.e., terms that do not contain variables. A term is linear, if it does not contain any variable more than once. A sorted map f from a sorted set A to a sorted set B, written $f : A \to B$, is a map such that $f(a) : \iota \in B$ whenever $a : \iota \in A$. A substitution is a sorted map $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F},\mathcal{V})$ for another sorted set $\mathcal{X}$ of variables,[2] and the instance is the term $t\sigma \in \mathcal{T}(\mathcal{F},\mathcal{V})$ obtained from $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ by replacing all x by $\sigma(x)$. We write $\sigma\delta$ for the composition of two substitutions σ and δ, and $[x \mapsto t]$ is the substitution which substitutes x by t and $y[x \mapsto t] = y$ for all $x \neq y$. A term $\ell : \iota \in \mathcal{T}(\mathcal{F},\mathcal{X})$ matches a term $t : \iota \in \mathcal{T}(\mathcal{F},\mathcal{V})$ if there exists a substitution $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F},\mathcal{V})$ such that $\ell\sigma = t$.

We consider programs that consist of a set of rules $\ell \to r$ and evaluation is defined by replacing instances of left-hand sides (lhss) $\ell\sigma$ by instances of right-hand sides $r\sigma$. For instance a program might be a TRS, or some other first-order functional programming language that uses pattern matching. We assume that there is a fixed finite signature $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{C}$ contains constructor symbols and $\mathcal{D}$ contains defined symbols. Hence, input values to a function are represented by constructor ground terms. We assume that there exists a constructor ground term $t : \iota \in \mathcal{T}(\mathcal{C})$ for each sort $\iota \in \mathcal{S}$. A sort ι is finite if $\{t \mid t : \iota \in \mathcal{T}(\mathcal{C})\}$ is a finite set, otherwise ι is infinite. We say a term $f(t_1,\dots,t_n)$ is *basic* if $f : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{D}$ and $t_1 : \iota_1, \dots, t_n : \iota_n \in \mathcal{T}(\mathcal{C},\mathcal{V})$, and denote the set of basic terms by $\mathcal{B}(\mathcal{C},\mathcal{D},\mathcal{V})$.

We are now ready to formally describe that evaluation of a program cannot get stuck.

▶ **Definition 2** (Pattern Completeness of Programs). *A program with lhss L is pattern complete, if every basic ground term $t \in \mathcal{B}(\mathcal{C},\mathcal{D},\emptyset)$ is matched by some $\ell \in L$.*

Pattern completeness is an instance of the *cover problem*, a notion that appears in the context of both term rewriting and functional programming [4, 11, 18]: a set L of terms *covers* a set T of terms if every constructor ground instance of a term in T is matched by some term in L. Clearly, pattern completeness of a program with lhss L is exactly the

[2] On paper it is not essential to distinguish the sets of variables, while it is convenient in the formalization that we can use variables different from those used to represent programs.

property that L covers $T := \{f(x_1,\dots,x_n) \mid f : \iota_1 \times \dots \times \iota_n \to \iota_0 \in \mathcal{D}\}$ where $x_1,\dots,x_n$ is a sequence of distinct variables; and the complement algorithm of Lazrek et al. [12] can be used to decide whether L covers T if all terms in L are linear.

An alternative notion to pattern completeness is quasi-reducibility [8], where the difference is that matching can happen for an arbitrary subterm.

▶ **Definition 3** (Quasi-Reducibility of Programs). *A program with lhss L is quasi-reducible, if every basic ground term $t \in \mathcal{B}(\mathcal{C},\mathcal{D},\emptyset)$ contains a subterm that is matched by some $\ell \in L$.*

Pattern completeness implies quasi-reducibility since t is a subterm of t, and the two notions coincide if the root symbols of all lhss are in $\mathcal{D}$ – as in the functional programming setting or in Example 1. Example 4 illustrates the difference between the two notions.

▶ **Example 4.** Consider $\mathcal{C}_\mathbb{Z} = \{\mathsf{true} : \mathbb{B}, \mathsf{false} : \mathbb{B}, \mathsf{0} : \mathbb{Z}, \mathsf{s} : \mathbb{Z} \to \mathbb{Z}, \mathsf{p} : \mathbb{Z} \to \mathbb{Z}\}$ to represent the Booleans and integers in a successor–predecessor notation, e.g., $\mathsf{p}(\mathsf{0})$ represents -1. Now we consider a TRS $\mathcal{R}_\mathbb{Z}$ that defines a function to compute whether an integer is even, i.e., $\mathcal{D} = \{\mathsf{even} : \mathbb{Z} \to \mathbb{B}\}$. It consists of all rules of $\mathcal{R}_\mathbb{N}$ in Example 1 and the following rules.

$$\mathsf{even}(\mathsf{p}(\mathsf{0})) \to \mathsf{false} \qquad \mathsf{even}(\mathsf{p}(\mathsf{p}(x))) \to \mathsf{even}(x) \tag{2}$$

$$\mathsf{s}(\mathsf{p}(x)) \to x \qquad \mathsf{p}(\mathsf{s}(x)) \to x \tag{3}$$

This TRS is quasi-reducible since every term $\mathsf{even}(n)$ with $n : \mathbb{Z} \in \mathcal{T}(\mathcal{C}_\mathbb{Z})$ has a subterm that is matched by some lhs: If n contains both s and p then one of the rules (3) is applicable. Otherwise n is of the form $\mathsf{s}^i(\mathsf{0})$ or $\mathsf{p}^i(\mathsf{0})$ and then rules (1) or (2) will be applicable.

The TRS is not pattern complete since $\mathsf{even}(\mathsf{s}(\mathsf{p}(\mathsf{0})))$ is not matched by any lhs.

3 Pattern Completeness – The Linear Case

Before we design the new decision procedure for pattern completeness we first reformulate and generalize this notion, leading to matching problems and pattern problems.

▶ **Definition 5** (Matching Problem and Pattern Problem). *A* matching problem *is a finite set $mp = \{(t_1,\ell_1),\dots,(t_n,\ell_n)\} \subseteq \mathcal{T}(\mathcal{F},\mathcal{V}) \times \mathcal{T}(\mathcal{F},\mathcal{X})$ of pairs of terms. A* pattern problem *is a finite set $pp = \{mp_1,\dots,mp_k\}$ of matching problems.*

A matching problem mp is complete *w.r.t. a constructor ground substitution $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C})$ if there is some substitution $\gamma : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ such that $t\sigma = \ell\gamma$ for all $(t,\ell) \in mp$. A pattern problem pp is complete if for each constructor ground substitution σ there is some $mp \in pp$ such that mp is complete w.r.t. σ. A set P of pattern problems is complete if each $pp \in P$ is complete.*

When expanding the definition of completeness of a set of pattern problems P we obtain an alternative definition, which reveals that there are two quantifier alternations.

$$P \text{ is complete iff } \forall pp \in P.\, \forall \sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C}).\, \exists mp \in pp.\, \exists \gamma : \mathcal{X} \to \mathcal{T}(\mathcal{F}).\, \forall (t,\ell) \in mp.\, t\sigma = \ell\gamma$$

Pattern problems are quite generic and can express several properties. For instance, L covers T iff the set of pattern problem $P = \{\{\{(t,\ell)\} \mid \ell \in L\} \mid t \in T\}$ is complete. Similarly, Aoto and Toyama's notion of strong quasi-reducibility [1] can also be encoded as a pattern problem: $pp = \{\{(t,\ell)\} \mid t \in \{x_1,\dots,x_n, f(x_1,\dots,x_n)\},\ \ell \in L\}$ expresses that one tries to find a match at the root ($t = f(x_1,\dots,x_n)$) or a match for a direct subterm ($t = x_i$). Finally, the question of whether a program with lhss L and defined symbols $\mathcal{D}$ is pattern complete w.r.t. Definition 2 is expressible as the completeness of the set of pattern problems $P = \{\{\{(f(x_1,\dots,x_n),\ell)\} \mid \ell \in L\} \mid f : \iota_1 \times \dots \times \iota_n \to \iota_0 \in \mathcal{D}\}$.

The following inference rules describe a decision procedure to determine the completeness of *linear* pattern problems. A matching problem $\{(t_1, \ell_1), \dots, (t_n, \ell_n)\}$ is linear if each ℓ_i is linear and $\mathsf{Var}(\ell_i) \cap \mathsf{Var}(\ell_j) = \emptyset$ for $i \neq j$. We say a pattern problem is *linear* if all its matching problems are linear.

In the inference rules we make use of a special matching problem $\bot_{mp}$ that represents an incomplete matching problem. Similarly, we define $\top_{pp}$ as a new pattern problem that is always complete. Finally, $\bot_P$ represents a new incomplete set of pattern problems.

▶ **Definition 6** (Inference Rules for Linear Pattern Problems). *We define $\to$ as the set of the following simplification rules for matching problems.*

$$\{(f(t_1,\dots,t_n), f(\ell_1,\dots,\ell_n))\} \uplus mp \to \{(t_1,\ell_1),\dots,(t_n,\ell_n)\} \cup mp \qquad \text{(decompose)}$$
$$\{(t,x)\} \uplus mp \to mp \qquad \textit{if } \forall (t',\ell) \in mp.\ x \notin \mathsf{Var}(\ell) \qquad \text{(match)}$$
$$\{(f(\dots), g(\dots))\} \uplus mp \to \bot_{mp} \qquad \textit{if } f \neq g \qquad \text{(clash)}$$

On top of this we define the set $\Rightarrow$ of simplification rules for pattern problems as follows:

$$\{mp\} \uplus pp \Rightarrow \{mp'\} \cup pp \qquad \textit{if } mp \to mp' \qquad \text{(simp-mp)}$$
$$\{\bot_{mp}\} \uplus pp \Rightarrow pp \qquad \text{(remove-mp)}$$
$$\{\emptyset\} \uplus pp \Rightarrow \top_{pp} \qquad \text{(success)}$$

Finally we provide rules $\Rrightarrow$ for modifying sets of pattern problems.

$$\{pp\} \uplus P \Rrightarrow \{pp'\} \cup P \qquad \textit{if } pp \Rightarrow pp' \qquad \text{(simp-pp)}$$
$$\{\emptyset\} \uplus P \Rrightarrow \bot_P \qquad \text{(failure)}$$
$$\{\top_{pp}\} \uplus P \Rrightarrow P \qquad \text{(remove-pp)}$$
$$\{pp\} \uplus P \Rrightarrow \mathsf{Inst}(pp, x) \cup P \qquad \textit{if } mp \in pp \textit{ and } (x, f(\dots)) \in mp \qquad \text{(instantiate)}$$

Here, for a pattern problem pp and a variable $x : \iota_0 \in \mathcal{V}$, the pattern problem set $\mathsf{Inst}(pp,x)$ consists of a pattern problem $pp\sigma_{x,c} = \{\{(t\sigma_{x,c}, \ell) \mid (t,\ell) \in mp\} \mid mp \in pp\}$ for each $c : \iota_1 \times \dots \times \iota_n \to \iota_0 \in \mathcal{C}$, where $\sigma_{x,c} = [x \mapsto c(x_1,\dots,x_n)]$ for distinct fresh variables $x_1 : \iota_1, \dots, x_n : \iota_n \in \mathcal{V}$.

Clearly, (decompose), (match) and (clash) correspond to a standard matching algorithm. Most of the other rules correspond to the universal and existential quantification that is done in the definition of completeness. The only exception is (instantiate). Here a matching algorithm would detect a failure since a variable x is never matched by a non-variable term $f(\dots)$. However, since the x in our setting just represents an arbitrary constructor ground term, we need to do case analysis on the outermost constructor. This is done by replacing $x : \iota_0 \in \mathcal{V}$ by all possible constructor terms of shape $c(x_1,\dots,x_n)$ for all $c : \iota_1 \times \dots \times \iota_n \to \iota_0 \in \mathcal{C}$.

The following theorem states that $\Rrightarrow$ can be used to decide completeness of linear pattern problems. Here, $\Rrightarrow^!$ is defined as reduction to normal form, i.e., $P \Rrightarrow^! P'$ iff $P \Rrightarrow^* P' \wedge \nexists P''.\ P' \Rrightarrow P''$.

▶ **Theorem 7** (Decision Procedure for Completeness of Linear Pattern Problems).
- *$\Rrightarrow$ is terminating.*
- *If $P \Rrightarrow P'$ then P is complete iff P' is complete.*
- *If P is linear and $P \Rrightarrow P'$, then P' is linear.*
- *If P is linear and $P \Rrightarrow^! P'$ then $P' \in \{\emptyset, \bot_P\}$.*
- *If P is linear, then P is complete iff $P \Rrightarrow^! \emptyset$.*

Proof. The property that normal forms will be either $\emptyset$ or $\bot_P$ follows by an easy analysis of the rules. In particular (clash), (decompose), (match), and (instantiate) cover all cases of a pair (t, ℓ) of a linear matching problem, i.e., whether these terms are variables or function applications, and whether the root symbol of both terms is identical or not. Note that the condition in (match) is always satisfied for linear matching problems.

Preserving completeness is rather obvious for all rules except (instantiate), which was already explained in the paragraph directly after Definition 6.

The most interesting aspect is termination. To prove it, we first define a measure of difference $|\ell - t|$ for a term pair (t, ℓ) of some matching problem:

- $|\ell - x|$ is the number of function symbols in ℓ,
- $|f(\ell_1, \ldots, \ell_n) - f(t_1, \ldots, t_n)| = \sum_{i=1}^{n} |\ell_i - t_i|$, and
- $|\ell - t| = 0$ in all other cases.

We lift this measure to pattern problems by $|pp|_{\text{diff}} = \sum_{mp \in pp, (t,\ell) \in mp} |\ell - t|$. Finally, we define $\succ$ as a relation on sets of pattern problems via the multiset extension $>^{mul}$ of $>$ by $P \succ P'$ iff $\{|pp|_{\text{diff}} \mid pp \in P\} >^{mul} \{|pp|_{\text{diff}} \mid pp \in P'\}$. This relation is strongly normalizing and has the nice property that each $\Rrightarrow$-step weakly decreases and the (instantiate) rule strictly decreases w.r.t. $\succ$. Hence, (instantiate) cannot be applied infinitely often. That the remaining rules terminate does not need a complicated measure: their application decreases the number of function symbols or the cardinalities of the sets. ◀

So, completeness of linear pattern problems is decidable. Let us illustrate the algorithm on a previous example.

▶ **Example 8.** The algorithm validates that $\mathcal{R}_\mathbb{N}$ in Example 1 is pattern complete.

$$\begin{aligned}
P = {} & \{\, \{\{(\mathsf{even}(y), \mathsf{even}(0))\}, \{(\mathsf{even}(y), \mathsf{even}(\mathsf{s}(0)))\}, \{(\mathsf{even}(y), \mathsf{even}(\mathsf{s}(\mathsf{s}(x))))\}\}\,\} \\
& \Rrightarrow^* \{\, \{\{(y, 0)\}, \{(y, \mathsf{s}(0))\}, \{(y, \mathsf{s}(\mathsf{s}(x)))\}\}\,\} \\
& \Rrightarrow \{\, \{\{(0,0)\}, \{(0, \mathsf{s}(0))\}, \{(0, \mathsf{s}(\mathsf{s}(x)))\}\},\ \{\{(\mathsf{s}(z), 0)\}, \{(\mathsf{s}(z), \mathsf{s}(0))\}, \{(\mathsf{s}(z), \mathsf{s}(\mathsf{s}(x)))\}\}\,\} \\
& \Rrightarrow^* \{\, \{\emptyset, \bot_{mp}, \bot_{mp}\},\ \{\bot_{mp}, \{(z, 0)\}, \{(z, \mathsf{s}(x))\}\}\,\} \\
& \Rrightarrow^* \{\, \{\{(z, 0)\}, \{(z, \mathsf{s}(x))\}\}\,\} \\
& \Rrightarrow \{\, \{\{(0,0)\}, \{(0, \mathsf{s}(x))\}\},\ \{\{(\mathsf{s}(y), 0)\}, \{(\mathsf{s}(y), \mathsf{s}(x))\}\}\,\} \\
& \Rrightarrow^* \{\, \{\emptyset, \bot_{mp}\},\ \{\bot_{mp}, \emptyset\}\,\} \\
& \Rrightarrow^* \emptyset
\end{aligned}$$

Regarding the complexity of the algorithm, one can prove an exponential upper bound on the number of $\Rrightarrow$-steps. Note that deciding quasi-reducibility is co-NP complete [7], and this result carries over to pattern completeness: the restriction to just search for matches at the root position in pattern completeness does not cause a lower complexity class. So unless P = NP, the exponential upper bound cannot be improved.

We briefly illustrate the idea of why pattern completeness is co-NP hard.

▶ **Theorem 9.** *Deciding pattern completeness is co-NP hard for both TRSs and pattern problems, even in the linear case.*

Proof. We perform a reduction from the Boolean satisfiability problem for conjunctive normal forms (CNFs). So assume φ is a CNF that contains n different Boolean variables $x_1, \ldots, x_n$ and consists of m clauses $c_1, \ldots, c_m$. W.l.o.g. we assume that no clause c_i contains conflicting literals, i.e., x_j and $\neg x_j$ for the same j, because such clauses are trivially valid and can therefore be removed from the set of clauses in polynomial time.

We translate φ into the following left-linear TRS. We use a signature that contains the two Booleans true and false as constructors, and there is one defined symbol $\mathsf{f} : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B}$ of arity n. We then define the TRS $\mathcal{R}_\varphi$ over this signature. It contains m rules where the ith rule has the form $\ell_i := \mathsf{f}(t_{i1}, \ldots, t_{in}) \to \mathsf{true}$ and t_{ij} is defined as false if $x_j \in c_i$, true if $\neg x_j \in c_i$, and x_j, otherwise. For instance, if $n = 5$ and $c_2 = \{\neg x_1, x_3, x_4\}$ then $\ell_2 = \mathsf{f}(\mathsf{true}, x_2, \mathsf{false}, \mathsf{false}, x_5)$. It is clear that this translation is computable in time proportional to $n \times m$.

For every variable assignment $\alpha : \{x_1, \ldots, x_n\} \to \mathbb{B}$, let f_α be the term $\mathsf{f}(\alpha(x_1), \ldots, \alpha(x_n))$. Consequently, for every α and clause c_i we see that $\alpha \models c_i$ iff f_α is not matched by ℓ_i. Therefore, φ is unsatisfiable iff $\mathcal{R}_\varphi$ is pattern complete.

Of course, instead of using the TRS $\mathcal{R}_\varphi$, one can equivalently use the linear pattern problem $pp_\varphi := \{\{(f(x_1, \ldots, x_n), \ell_1)\}, \ldots, \{(f(x_1, \ldots, x_n), \ell_m)\}\}$ and derive that φ is unsatisfiable iff pp_φ is complete. ◀

4 Pattern Completeness – The General Case

In the non-linear case $\Rrightarrow$ might get stuck, e.g., if there is a matching problem $\{(t, x), (t', x)\}$ for $t \neq t'$. To treat such cases we have to add further simplification rules. In order to do so without breaking termination, we need to distinguish between finite and infinite sorts. To illustrate the problem, consider a TRS to determine the majority of three values: $\{f(x, x, y) \to x, f(x, y, x) \to x, f(y, x, x) \to x\}$. If x is a variable of a finite sort, then the lhss can be pattern complete: if the sort allows at most two different values, such as the Booleans, then the lhss cover all cases. If the sort has at least three values a, b, c, then no lhs matches $f(a, b, c)$. So, we want to allow instantiating variables to judge pattern completeness, but we cannot allow instantiating variables of infinite sorts, since otherwise the resulting inference rules would no longer be terminating.

As final preparation for the new inference rules we define (the only two) reasons on why two terms differ.

▶ **Definition 10.** *We say that terms t and t'* clash *if $t|_p = f(\ldots) \neq g(\ldots) = t'|_p$ with $f \neq g$ for some shared position p of t and t'. The terms t and t'* differ in variable x *if $t|_p \neq t'|_p$ and $x \in \{t|_p, t'|_p\}$ for some shared position p.*

▶ **Definition 11** (Inference Rules for General Pattern Problems). *We take all rules of Definition 6 and add the following ones.*

$$\{(t, x), (t', x)\} \uplus mp \to \bot_{mp} \qquad \text{if } t \text{ and } t' \text{ clash} \qquad \text{(clash')}$$

$$\{pp\} \uplus P \Rrightarrow \mathsf{Inst}(pp, x) \cup P \qquad \text{(instantiate')}$$

if $mp \in pp$, $\{(t, y), (t', y)\} \subseteq mp$, t and t' differ in variable $x : \iota \in \mathcal{V}$, and ι is finite

$$\{pp\} \uplus P \Rrightarrow \bot_P \qquad \text{if for each } mp \in pp \text{ there are } \{(t, y), (t', y)\} \subseteq mp \qquad \text{(failure')}$$

such that t and t' differ in variable $x : \iota \in \mathcal{V}$ and ι is infinite

Indeed, with these new rules, $\Rrightarrow$ cannot get stuck even for non-linear inputs.

We first remark that there is a different flavor of problems with non-linear matching problems of the form $\{(t, x), (t', x)\}$. A clash of t and t' can always be resolved locally by (clash'). If there is a difference in a finite-sort variable, this can also be handled locally by (instantiate'). However, differences in infinite-sort variables can only be applied via (failure') if indeed all matching problems show such a difference. Note that it is unsound to turn (failure') into a local rule for matching problems, i.e., if we would make (failure') similar to (clash').

▶ **Example 12.** Consider lhss $\{f(x,x), f(\mathsf{s}(x),y), f(x,\mathsf{s}(y))\}$, where $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is defined and $\mathsf{0} : \mathbb{N}$ and $\mathsf{s} : \mathbb{N} \to \mathbb{N}$ are the constructors. We can start to apply the rules as follows:

$$\begin{aligned}&\{\{(f(x_1,x_2), f(x,x))\}, \{(f(x_1,x_2), f(\mathsf{s}(x),y))\}, \{(f(x_1,x_2), f(x,\mathsf{s}(y)))\}\}\\ \Rightarrow^* &\{\{(x_1,x),(x_2,x)\}, \{(x_1,\mathsf{s}(x))\}, \{(x_2,\mathsf{s}(y))\}\} =: pp\end{aligned}$$

If at this point we would remove the matching problem $\{(x_1,x),(x_2,x)\}$ from pp because of a difference in variable x_1, then we would switch from the complete pattern problem pp into the incomplete pattern problem $pp \setminus \{\{(x_1,x),(x_2,x)\}\}$: the constructor ground substitution σ where $\sigma(x_1) = \sigma(x_2) = \mathsf{0}$ is covered by pp, but not by $pp \setminus \{\{(x_1,x),(x_2,x)\}\}$.

We state a similar theorem to the linear case, though its proof is much more evolved.

▶ **Theorem 13** (Decision Procedure for Completeness of Pattern Problems). *Consider $\Rrightarrow$ of Definition 11.*

- *$\Rrightarrow$ is terminating.*
- *If $P \Rrightarrow^! P'$ then $P' \in \{\emptyset, \bot_P\}$.*
- *If $P \Rrightarrow P'$ then P is complete iff P' is complete.*
- *P is complete iff $P \Rrightarrow^! \emptyset$.*

Proof. Not getting stuck is the easiest of these properties and we leave it as an exercise to the interested reader to show that the only possible normal forms are $\emptyset$ and $\bot_P$.

Termination is more complicated. We reuse the difference measure $|\cdot|_{\text{diff}}$ from the linear case, and combine it with three other measures:

- $|pp|_{\text{symbols}}$ counts the number of all function symbols within pp.
- $|pp|_{\text{set-size}}$ sums up the cardinalities of all sets, while ignoring the term structure.
- $|pp|_{\text{fin-vars}}$ is used to measure the size of variables of finite sort. To be more precise, for a variable x having some finite sort ι, we first define $|x|$ as the maximal term size of a constructor ground term of sort ι. On top of this auxiliary measure we define

  $$|pp|_{\text{fin-vars}} = \sum_{x \in \{x.\ mp \in pp, (t,\ell) \in mp, x:\iota \in \mathsf{Var}(t), \iota \text{ is finite}\}} |x|$$

  In other words, for $|\cdot|_{\text{fin-vars}}$ we collect all variables of finite sort and sum up the maximal term sizes.

The order $\succ$ on pattern problems is then defined as the lexicographic comparison of the quadruples $(|pp|_{\text{diff}}, |pp|_{\text{fin-vars}}, |pp|_{\text{symbols}}, |pp|_{\text{set-size}})$ that are obtained from a pattern problem pp. Multisets of pattern problems P are compared by $\succ^{mul}$.

All rules on sets of pattern problems give rise to a decrease w.r.t. $\succ^{mul}$, as it is indicated in the following matrix:

	$\lvert\cdot\rvert_{\text{diff}}$	$\lvert\cdot\rvert_{\text{fin-vars}}$	$\lvert\cdot\rvert_{\text{symbols}}$	$\lvert\cdot\rvert_{\text{set-size}}$
(instantiate)	$>$	$-$	$-$	$-$
(instantiate')	$\geq$	$>$	$-$	$-$
(simp-pp)	$\geq$	$\geq$	$\geq$ or $>$	$>$ or $-$
(remove-pp)	$\geq$	$\geq$	$\geq$	$>$

where the decrease in (simp-pp) depends on underlying applied rules. For instance, (decompose) leads to a decrease w.r.t. $|\cdot|_{\text{symbols}}$, but might increase $|\cdot|_{\text{set-size}}$, whereas (remove-mp) does not change $|\cdot|_{\text{symbols}}$, but gives a decrease w.r.t. $|\cdot|_{\text{set-size}}$. The reason for the decrease of (instantiate') is that whenever we instantiate some variable x of finite sort ι by $c(x_1,\dots,x_n)$, then $|x| \geq 1 + |x_1| + \ldots + |x_n| > |x_1| + \ldots + |x_n|$, and hence the $|\cdot|_{\text{fin-vars}}$-measure strictly decreases.

Partial correctness is the most challenging part, where the difficulty is the new rule (failure'). We prove that pp is not complete, whenever (failure') is applied for pp. To this end, we build a constructor ground substitution σ such that $pp\sigma$ contains a clash.

In detail: We use an invariant φ on pattern problems defined as follows: $\varphi(pp)$ iff for each $mp \in pp$ there are y, t, t' such that $\{(t, y), (t', y)\} \subseteq mp$, t and t' contain no variable of finite sort, and moreover these terms clash or differ in a variable of infinite sort. Suppose that (failure') is applicable for a pattern problem pp. To initially satisfy φ, we apply an initial partial ground substitution σ_0 on pp, where all variables of finite sort are instantiated by some constructor ground term of that sort. Since (failure') is applicable, we know that all matching problems in $pp\sigma_0$ still have a difference in a variable of infinite sort, since none of these variables gets instantiated by σ_0.

Having that $pp\sigma_0$ satisfies the invariant φ, we now iteratively remove all variables of infinite sort that cause a difference. To be more precise, whenever some t and t' differ in a variable x in $pp\sigma_i$, we build a constructor ground term u that is larger than any of the terms in $pp\sigma_i$. Note that such u exists, since x has an infinite sort. We let σ_{i+1} to be $\sigma_i[x \mapsto u]$. This process will terminate in finitely many steps, since the set of variables in $pp\sigma_{i+1}$ is a strict subset of that of $pp\sigma_i$.

At the end of the process, we obtain a partial constructor ground substitution σ_n that instantiates all variables that contribute to differences into ground terms. Assuming that φ is maintained in the process, it is easy to see that the final pattern problem is not complete: all matching problems of the final pattern problem contain a clash of some terms t and t', i.e., we can extend σ_n to any constructor ground substitution σ that instantiates the remaining variables.

Finally we prove that the step from $pp\sigma_i$ to $pp\sigma_{i+1}$ indeed preserves φ. To this end, pick any terms t and t' of matching problem $mp \in pp\sigma_i$ that cause the conflict within the invariant. We show by case analysis that also $t[x \mapsto u]$ and $t'[x \mapsto u]$ result in a conflict.

- if t and t' clash at position p, then there also is a clash of $t[x \mapsto u]$ and $t'[x \mapsto u]$ at the same position p;
- if $t|_p = y \neq t'|_p$ for some $y \neq x$, then $t|_p[x \mapsto u] = y \neq t'|_p[x \mapsto u]$ shows that again there is a difference in y;
- if $t|_p = x \neq t'|_p$ and $t'|_p$ contains a variable $y \neq x$, then $t|_p[x \mapsto u] = u$ is a ground term and thus must have a difference with $t'|_p[x \mapsto u]$ in the variable y;
- if $t|_p = x \neq t'|_p$ and $t'|_p$ is a ground term, then $t|_p[x \mapsto u] = u \neq t'|_p = t'|_p[x \mapsto u]$, since u is strictly larger than any term of $pp\sigma_i$, and therefore there must be a clash; and finally
- if $t|_p = x \neq t'|_p$ and $\mathsf{Var}(t'|_p) = \{x\}$, then $t|_p[x \mapsto u] = u$ and $t'|_p[x \mapsto u]$ contains u as a strict subterm. Hence $t|_p[x \mapsto u]$ and $t'|_p[x \mapsto u]$ are two different ground terms which must contain a clash. ◀

Note that $\Rrightarrow$ can be modified to support counter-example generation, i.e., instead of $\bot_P$ one returns a constructor substitution that violates the completeness of a set of pattern problems. To achieve this, for each pattern problem we additionally store a constructor substitution δ. This substitution is initially empty, i.e., nothing is instantiated, and δ will keep track of the instantiations that are performed during the algorithm. The inference rules of $\Rrightarrow$ are modified as follows

$$\{(pp, \delta)\} \uplus P \Rrightarrow \{(pp', \delta)\} \cup P \quad \text{if } pp \Rightarrow pp' \qquad \text{(simp-pp)}$$

$$\{(\top_{pp}, \delta)\} \uplus P \Rrightarrow P \qquad \text{(remove-pp)}$$

$$\{(\emptyset, \delta)\} \uplus P \Rrightarrow \delta \qquad \text{(failure)}$$

$$\{(pp, \delta)\} \uplus P \Rrightarrow \{(pp\sigma_{x,c}, \delta\sigma_{x,c}) \mid c : \dots \in \mathcal{C}\} \cup P \quad \text{if } \dots \qquad \text{(instantiate('))}$$

$$\{(pp, \delta)\} \uplus P \Rrightarrow \delta\sigma \quad \text{if } \dots \qquad \text{(failure')}$$

where the dots refer to the conditions of the original rules, and in (failure') the substitution σ is defined as the constructor ground substitution that is described in the partial correctness proof of the (failure') rule in Theorem 13.

5 Formalization and Implementation

In the formalization we first describe our development on sorted term rewriting (Section 5.1, 1706 lines of Isabelle), which is essential to develop the formalization of the decision procedure of Section 4 in Isabelle. The latter uses three different layers of abstraction. We start with an abstract set based formalization (Section 5.2, 1125 lines), then refine it to a version based on multisets (Section 5.3, 828 lines), and finally arrive at an executable version that uses lists to represent matching and pattern problems (Section 5.4, 1322 lines).

5.1 Sorted Term Rewriting

Here we present our formalization of sorted term rewriting. First, we reuse the datatype for terms from the AFP entry First-Order Terms [17].

datatype $('f, 'v)$ *term* $=$ *Var* $'v$ | *Fun* $'f$ $(('f, 'v)$ *term list*$)$

This datatype collects all unsorted, variadic terms. Next we introduce a sorted set over the datatype $('f, 'v)$ *term*. We characterize a sorted set as a partial map that assigns an element a sort. It is partial, in the sense that unsorted elements are not assigned a sort. Partial maps are readily supported in Isabelle/HOL as types of form $'a \rightharpoonup\ 's$, which is a synonym of $'a \to\ 's$ *option*. We just introduce the notation "$a : \iota$ *in* A" to mean that a sorted set $A :: 'a \rightharpoonup\ 's$ assigns its element $a :: 'a$ a sort $\iota :: 's$.

definition ... **where** $a : \iota\ \mathit{in}\ A \equiv A\ a = \mathit{Some}\ \iota$

Hereafter, we often omit Isabelle specifications for introducing notations by "...".

We formalize sorted signatures also as partial maps:

type_synonym $('f, 's)$ *ssig* $=\ 'f \times\ 's\ \mathit{list} \rightharpoonup\ 's$

and introduce the following notation:

definition ... **where** $f : \iota s \to \iota_0\ \mathit{in}\ F \equiv F\ (f, \iota s) = \mathit{Some}\ \iota_0$

Given a sorted signature $F :: ('f, 's)$ *ssig* and a sorted set $V :: 'v \rightharpoonup\ 's$ of variables, we define the sorted set $\mathcal{T}(F, V) :: ('f, 'v)$ *term* $\rightharpoonup\ 's$ of terms so that

$$\mathit{Var}\ v : \iota\ \mathit{in}\ \mathcal{T}(F,V) \iff v : \iota\ \mathit{in}\ V$$
$$\mathit{Fun}\ f\ ts : \iota_0\ \mathit{in}\ \mathcal{T}(F,V) \iff (\exists \iota s.\ f : \iota s \to \iota_0\ \mathit{in}\ F \wedge ts :_l \iota s\ \mathit{in}\ \mathcal{T}(F,V))$$

Here, $as :_l \iota s\ \mathit{in}\ A$ denotes that the lists as and ιs have the same length and the ith element of as has the ith sort of ιs. We also introduce the notation $\emptyset$ for *Map.empty*, the partial map such that $\emptyset\ a = \mathit{None}$ for any a. Then $\mathcal{T}(F,\emptyset)$ represents the sorted set of ground terms. Given two signatures C and D, the set of basic terms is formalized as follows:

definition ... **where**

$$\mathcal{B}(C,D,V) = \{\mathit{Fun}\ f\ ts \mid f\ \iota s\ \iota_0\ ts\ .\ f : \iota s \to \iota_0\ \mathit{in}\ D \wedge ts :_l \iota s\ \mathit{in}\ \mathcal{T}(C,V)\}$$

A *sorted map* from a sorted set A to a sorted set B, written $f :_s A \to B$ in Isabelle, is a map f such that $a : \iota\ in\ A \implies f\ a : \iota\ in\ B$. In particular, sorted maps of form $\sigma :_s X \to \mathcal{T}(F,V)$ are the sorted substitutions. The application of a substitution σ on a term t is already defined as $t \cdot \sigma$ in the library in the unsorted setting. We additionally provide facts such as

lemma *subst_hastype*: $\sigma :_s X \to \mathcal{T}(F,V) \implies t : \iota\ in\ \mathcal{T}(F,X) \implies t \cdot \sigma : \iota\ in\ \mathcal{T}(F,V)$

The formalization of when a term matches another is straightforward:

definition ... **where** $l\ matches\ t = (\exists\ \sigma.\ t = l \cdot \sigma)$

and now we are ready to define the pattern completeness.

definition ... **where**
$pat\_complete\_lhss\ C\ D\ L = (\forall\ t \in \mathcal{B}(C,D,\emptyset).\ \exists l \in L.\ l\ matches\ t)$

5.2 Formalization of the Algorithm – Set Layer

The set based formalization is the one that is the furthest away from an executable version. Interestingly, it also deviates quite a bit from the textual description of the algorithm. Still, it is useful for proving that completeness of pattern problems is not altered by $\Rrightarrow$.

There are some deviations from the textual description that we like to mention.

First, we do not introduce the special problems $\bot_{mp}$ and $\top_{pp}$, e.g., by using an option-type. Instead, we split each set of inference rules in two parts, e.g., the matching problem transformation relation $\to$ into relation $\to_s$ that modifies an existing problem and into predicate *mp_fail* that leads to the special problem $\bot_{mp}$. In this way, the representation of matching and pattern problems stays simple, i.e., they are just (sets of) sets of pairs of terms.

Second, we change all $\uplus$-operators in the textual description into $\cup$-operators. This simplifies the reasoning for the refinements in the upcoming layers, but introduces nontermination. For instance, if $P = \{\top_{pp}\}$ then $P = \{\top_{pp}\} \cup P \Rrightarrow P$. Giving up on termination at this layer, we also join (instantiate) and (instantiate'): the formalization contains only one rule for instantiation at this layer, and this rule has no side-condition; i.e., it is always possible to instantiate $\{pp\}$ by $\mathsf{Inst}(pp, x)$ for any x.

Third, the formalization contains a notion of well-formedness for matching and pattern problems. In detail, the algorithm is formulated within a context that fixes a set S of sorts. Well-formedness enforces that the variables that occur in the problems only use sorts in S. Many of the properties are only proven for well-formed problems, and it is additionally proven that well-formedness is preserved by the transformations. Well-formedness does not enforce that the sets in a problem are finite; this is another source of nontermination on this layer.

We provide some example Isabelle snippets that formalize the relations $\to$ and $\Rightarrow$, illustrating the first two kinds of deviations. Here *insert a A* is Isabelle's notation for $\{a\} \cup A$.

inductive ... **where** $mp \to_s mp$
| $length\ ts = length\ ls \implies insert\ (Fun\ f\ ts,\ Fun\ f\ ls)\ mp \to_s set\ (zip\ ts\ ls) \cup mp$
| $x \notin \bigcup\ (vars\ `\ snd\ `\ mp) \implies insert\ (t,\ Var\ x)\ mp \to_s mp$

inductive *mp_fail* :: $('f,'v,'s)match\_problem\_set \to bool$
where $(f,length\ ts) \neq (g,length\ ls) \implies mp\_fail\ (insert\ (Fun\ f\ ts,\ Fun\ g\ ls)\ mp)$
| `(* further inference rule for clash' *)`

inductive … **where** $mp \rightarrow_s mp' \Longrightarrow$ *insert* $mp\ pp \Rightarrow_s$ *insert* $mp'\ pp$
| *mp_fail* $mp \Longrightarrow$ *insert* $mp\ pp \Rightarrow_s pp$

The relation $\Rrightarrow$ is formalized similarly. The main result of this layer is that $\Rrightarrow$ preserves pattern completeness on well-formed pattern problem sets, in Isabelle, *wf_pats*.

theorem *P_step_set_pcorrect*:
  $P \Rrightarrow_s P' \Longrightarrow$ *wf_pats* $P \Longrightarrow$ *pats_complete* $P \longleftrightarrow$ *pats_complete* P'

The most challenging rule was (failure') as detailed in the previous section. On the other hand, the most tedious one was (instantiate), which looks rather obvious on paper, but required 140 lines in our formalization.

5.3 Formalization of the Algorithm – Multiset Layer

On the next layer we use finite multisets to represent the algorithm. This layer is the one that is closest to the textual description and we fully prove Theorem 13 for this representation. The design of the formalization is as follows.

Concerning the relationship between textual and formalized version of the algorithm, we keep the deviation of splitting the inference rules from the previous layer, so that there is no need for the special problems $\bot_{mp}$ and $\top_{pp}$. Since a multiset union operation corresponds to a $\uplus$-operation on sets, there is no deviation at this point anymore. However, we require one further inference rule for matching problems whose necessity does not arise when working with sets. Since a multiset can have multiple occurrences of the same element, we need an explicit inference rule that is capable of deleting duplicates. To this end, we add the rule

$$\{(t,\ell),(t,\ell)\} \cup mp \rightarrow \{(t,\ell)\} \cup mp \qquad \text{(duplicate)}$$

on the multiset layer, which is then simulated by a new identity rule $mp \rightarrow mp$ on the set layer.

Partial correctness of $\Rrightarrow$ on this layer is obtained via the partial correctness result of $\Rrightarrow$ from the previous layer by proving a refinement property: The multiset-based implementation can be simulated by the set-based one.

The major new property that is added on this level is a formal proof of termination by closely following the textual proof.

We arrive at a formal version of Theorem 13 that looks quite similar to the textual one. Here, $\Rrightarrow$ in Isabelle refers to the multiset representation of $\Rrightarrow$, *SN* is strong normalization, i.e., termination, *pats_mset* converts from the multiset representation of pattern problems into the set representation, $\{\#\}$ is the empty multiset, and *bottom_mset* is representing $\bot_P$ as the multiset $\{\emptyset\}$.

theorem *SN_P_step*: *SN* $\Rrightarrow$
theorem *P_step*:
  assumes *wf_pats* (*pats_mset* P) **and** $(P,Q) \in \Rrightarrow^!$
  shows $Q = \{\#\} \wedge$ *pats_complete* (*pats_mset* P)
    $\vee\ Q =$ *bottom_mset* $\wedge \neg$ *pats_complete* (*pats_mset* P)

5.4 Formalization of the Algorithm – List Layer

In the final layer we provide an executable version of the algorithm, by refining the multiset based version. To this end, we switch from multisets to lists; we turn the inductive inference rules into a recursive function definition; and we specify the order in which the inference rules will be applied. Our list-based implementation is split into several phases.

In the first phase, we exhaustively apply rules (decompose), (duplicate), (clash) and (clash'). Moreover, we organize the representation of the matching problems as follows.

- We store a list of pairs that have the structure $(x, f(\dots))$, i.e., those pairs on which (instantiate) is applicable.
- We store another list of pairs $([t_1, \dots, t_n], x)$ such that the matching problem contains all pairs $(t_1, x), \dots, (t_n, x)$, and no combination t_i and t_j results in a clash, and the list $[t_1, \dots, t_n]$ is distinct.
- We further store a Boolean flag whether the matching problem satisfies the condition of (instantiate') or not.

In the second phase, we apply rule (match) exhaustively and try to apply (failure'). Both of these steps can efficiently be implemented based on the previously described representation of matching problems.

Finally, if nothing else is applicable, then in the third phase we invoke (instantiate) or (instantiate'), with a preference on the former. In order to create fresh variables for the application of these rules, we assume that these variables are just numbers, and use a global index n which is incremented whenever a fresh variable is required.

These three phases are then iterated in a recursive function until a normal form is reached. By induction on $\Rrightarrow$, it is shown that the list-based implementation refines the multiset version of $\Rrightarrow$. Hence, partial correctness is easily transferred from the previous layer.

There is some additional glue-code required to get the final algorithm.

- We need to compute a high-enough value for the initial variable index n.
- We need to check the prerequisite that was stated at the beginning of this paper, namely that indeed all sorts are inhabited: $\{t \mid t : \iota \in \mathcal{T}(\mathcal{C})\} \neq \emptyset$. To this end, we verify a standard marking algorithm that computes the set of inhabited sorts: initially no sort is marked as inhabited, and whenever $c : \iota_1 \times \dots \times \iota_n \to \iota_0 \in \mathcal{C}$ is a constructor and all sorts $\iota_1, \dots, \iota_n$ are marked, then also ι_0 is marked as inhabited. Finally, exactly the inhabited sorts are marked.
- We further require a function that determines whether a sort is finite or infinite; also here we verify a marking algorithm: initially no sort is marked as being finite, and whenever there is a sort ι where all constructors of that sort only have input sorts that are marked, then also ι is marked as finite. Finally, exactly the finite sorts are marked.[3] Interestingly, the dual approach (marking of infinite sorts whenever a recursive constructor is detected) is not so straight-forward, because sorts might be mutual recursive without direct recursion.

We finally provide a few wrapper functions that invoke the main decision procedure and get rid of its preconditions. For instance, for pattern completeness of programs (represented by their lhss) we obtain an algorithm *decide_pat_complete_lhss*.

theorem *decide_pat_complete_lhss*:
  assumes *decide_pat_complete_lhss C D lhss = return b*
  shows *b = pat_complete_lhss (map_of C) (map_of D) (set lhss)*

The algorithm will report an error on invalid input, e.g., if not all sorts are inhabited, or if the list of constructors C or the list of defined symbols D contain conflicting sort informations. If no such error is reported then the return value will be a Boolean b, and b is the completeness property of the set of lhss.

[3] The Isabelle formalization actually slightly deviates from this representation. It starts from the set of potentially infinite sorts and then iteratively removes the finite sorts.

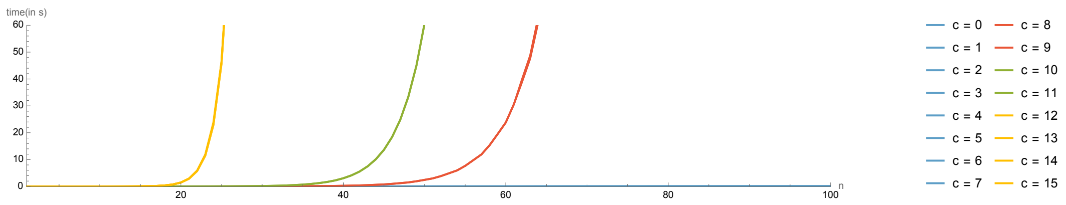

Figure 1 Timing of our algorithm for each configuration $c \in \{0, \ldots, 15\}$ on different sizes $n \in \{1, \ldots, 100\}$ using a timeout of 60 seconds.

6 Experiments

In order to evaluate the efficiency of our decision procedure, we use the following set of TRSs. They are similar to test programs that are used to show exponential behavior of match compilers for functional programming languages [16, Example 6].

▶ **Example 14.** We define TRS $\mathcal{R}_{c,n}$ for different configurations $c \in \{0, \ldots, 15\}$ and different sizes n. All TRSs use only one sort, namely the Booleans with constructors T and F, and there is only one defined symbol g. The TRS $\mathcal{R}_{c,n}$ consists of $2n + 1$ many rules and g has arity $2n$. We do not provide a full formal definition of $\mathcal{R}_{c,n}$, but instead illustrate the lhss of $\mathcal{R}_{c,n}$ for $c \in \{0, 1, 2, 4\}$ and $n = 3$, where each occurrence of _ represents a fresh variable.

$c = 0$	$c = 1$	$c = 2$	$c = 4$
$\mathsf{g}(\mathsf{F}, \_, \mathsf{F}, \_, \mathsf{F}, \_)$	$\mathsf{g}(\mathsf{F}, \mathsf{F}, \mathsf{F}, \_, \_, \_)$	$\mathsf{g}(\mathsf{F}, \_, \mathsf{F}, \_, \mathsf{F}, \_)$	$\mathsf{g}(\mathsf{T}, \mathsf{T}, \_, \_, \_, \_)$
$\mathsf{g}(\mathsf{T}, \mathsf{T}, \_, \_, \_, \_)$	$\mathsf{g}(\mathsf{T}, \_, \_, \mathsf{T}, \_, \_)$	$\mathsf{g}(\mathsf{T}, \mathsf{T}, \_, \_, \_, \_)$	$\mathsf{g}(\mathsf{T}, \mathsf{F}, \_, \_, \_, \_)$
$\mathsf{g}(\mathsf{T}, \mathsf{F}, \_, \_, \_, \_)$	$\mathsf{g}(\mathsf{T}, \_, \_, \mathsf{F}, \_, \_)$	$\mathsf{g}(\_, \_, \mathsf{T}, \mathsf{T}, \_, \_)$	$\mathsf{g}(\_, \_, \mathsf{T}, \mathsf{T}, \_, \_)$
$\mathsf{g}(\_, \_, \mathsf{T}, \mathsf{T}, \_, \_)$	$\mathsf{g}(\_, \mathsf{T}, \_, \_, \mathsf{T}, \_)$	$\mathsf{g}(\_, \_, \_, \_, \mathsf{T}, \mathsf{T})$	$\mathsf{g}(\_, \_, \mathsf{T}, \mathsf{F}, \_, \_)$
$\mathsf{g}(\_, \_, \mathsf{T}, \mathsf{F}, \_, \_)$	$\mathsf{g}(\_, \mathsf{T}, \_, \_, \mathsf{F}, \_)$	$\mathsf{g}(\mathsf{T}, \mathsf{F}, \_, \_, \_, \_)$	$\mathsf{g}(\_, \_, \_, \_, \mathsf{T}, \mathsf{T})$
$\mathsf{g}(\_, \_, \_, \_, \mathsf{T}, \mathsf{T})$	$\mathsf{g}(\_, \_, \mathsf{T}, \_, \_, \mathsf{T})$	$\mathsf{g}(\_, \_, \mathsf{T}, \mathsf{F}, \_, \_)$	$\mathsf{g}(\_, \_, \_, \_, \mathsf{T}, \mathsf{F})$
$\mathsf{g}(\_, \_, \_, \_, \mathsf{T}, \mathsf{F})$	$\mathsf{g}(\_, \_, \mathsf{T}, \_, \_, \mathsf{F})$	$\mathsf{g}(\_, \_, \_, \_, \mathsf{T}, \mathsf{F})$	$\mathsf{g}(\mathsf{F}, \_, \mathsf{F}, \_, \mathsf{F}, \_)$

The 16 configurations are obtained by combining 4 different kinds to arrange the arguments of g with 4 different orders of the rules. For the argument orders of g we choose the following four alternatives, visualized by reordering the arguments of the first lhs of $\mathcal{R}_{0,n}$: $\mathsf{g}(\mathsf{F}, \_, \mathsf{F}, \_, \ldots, \mathsf{F}, \_)$ or $\mathsf{g}(\_, \mathsf{F}, \_, \mathsf{F}, \ldots, \_, \mathsf{F})$ or $\mathsf{g}(\mathsf{F}, \mathsf{F}, \ldots, \mathsf{F}, \_, \_, \ldots, \_)$ or $\mathsf{g}(\_, \_, \ldots, \_, \mathsf{F}, \mathsf{F}, \ldots, \mathsf{F})$. Concerning the order of the rules, we either put the first rule of $\mathcal{R}_{0,n}$ to the front position ($c = 0$) or to the last position ($c = 4$); and we either group the other $2n$ rules in n blocks of size 2 ($c = 0$) or in 2 blocks of size n ($c = 2$).

Since for a given n all configurations result in the same set of lhss (modulo symmetries), the question of pattern completeness should be equally hard for all configurations. However, since our implementation of the decision procedure has a fixed order in which rules are applied and in which variables are instantiated, there is quite a different behavior in the execution time, cf. Figure 1.

Choosing $c \in \{0, \ldots, 7\}$ results in a low execution time, where the corresponding blue line in Figure 1 is not distinguishable from the x-axis: for instance, deciding pattern completeness of $\mathcal{R}_{c,100}$ is finished within 0.06 seconds. However, for $c \in \{8, \ldots, 15\}$ an exponential behavior becomes visible, where $c = \{8, 9\}$, $c = \{10, 11\}$, and $c = \{12, 13, 14, 15\}$ each have similar behavior.

We further compare our decision procedure with three other algorithms.

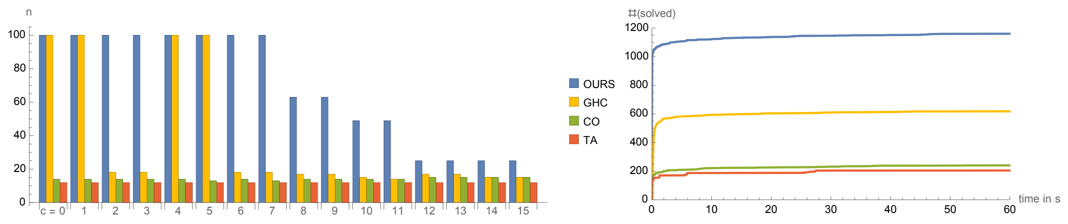

Figure 2 Left: Increase n for each $c \in \{0, \dots, 15\}$ until $n = 100$ or 60 seconds timeout is reached. Right: Number of solved $\mathcal{R}_{c,n}$ instances for $(c, n) \in \{0, \dots, 15\} \times \{1, \dots, 100\}$ within time limit.

- GHC: we encode $\mathcal{R}_{c,n}$ as a Haskell program and use the `ghc` Haskell compiler and ask it to warn about incomplete patterns. To be more precise we invoke `ghc` with parameters `-c -Wincomplete-patterns -fmax-pmcheck-models=1000000` where the latter number is chosen in such a way that no approximation is occurring.[4]
- CO: we run the complement algorithm on the TRSs, taking the implementation that is available in the ground confluence prover AGCP [1].
- TA: given $\mathcal{R}_{c,n}$ we define two tree automata $\mathcal{A}_n$ (with transitions $\mathsf{F} \to \mathsf{bool}$, $\mathsf{T} \to \mathsf{bool}$ and $\mathsf{g}(\mathsf{bool}, \dots, \mathsf{bool}) \to \mathsf{accept}$) and $\mathcal{B}_{c,n}$ (using four common transitions $\mathsf{F} \to \mathsf{bool}$, $\mathsf{F} \to \mathsf{false}$, $\mathsf{T} \to \mathsf{bool}$, $\mathsf{T} \to \mathsf{true}$, and one further transition for each rule, e.g., $\mathsf{g}(\mathsf{bool}, \mathsf{bool}, \mathsf{bool}, \mathsf{bool}, \mathsf{true}, \mathsf{false}) \to \mathsf{accept}$ for the last rule of $\mathcal{R}_{0,3}$) so that pattern completeness of $\mathcal{R}_{c,n}$ is equivalent to the language inclusion problem $\mathcal{L}(\mathcal{A}_n) \subseteq \mathcal{L}(\mathcal{B}_{c,n})$. We then invoke the tree automaton library of FORT-h [13] to decide this inclusion property.

We ran experiments where for each configuration c and each algorithm we increased n until either $\mathcal{R}_{c,n}$ for $n = 100$ was successfully analyzed, or until there was a 60 seconds timeout when handling $\mathcal{R}_{c,n}$. In Figure 2 we display the maximal values of n (left) and the cumulative solved instances plot in the style of SAT-competition [6] (right).

The diagram clearly shows that our new decision procedure outperforms all other three algorithms on the test suite. Interestingly, also in GHC there is a strong dependence on the configuration, i.e., the execution time varies between polynomial and exponential. This is different for TA and CO: these algorithms always resulted in exponential behavior, independent of the choice of c.

For further details on the experiments we refer to the website with supplementary material.

7 Conclusion and Future Work

We developed a new decision procedure to decide pattern completeness that is not restricted to the left-linear case. The corresponding verified implementation is faster than previous approaches, in particular it performs better than the complement algorithm and tree automata based methods.

We see some opportunities for future work. First, one can integrate an improved strategy to select variables for instantiation, in particular since permutations in the input cause severe differences in runtime. One can also try to further improve the implementation, e.g., by

[4] When invoking `ghc`, it does not only check pattern completeness, but also compiles the program. However, the compilation time is negligible in our experiments. On all programs where GHC was successful, the compilation time decreased to below 0.5 seconds when turning off the pattern completeness check.

following suggestions of Sestoft [16, Section 7.5] such as the integration of memoization. The latter corresponds to a rule $\{pp, pp\} \cup P \Rrightarrow \{pp\} \cup P$ to detect and eliminate duplicate pattern problems. Second, one might add counter-example generation into the formalization and into the verified implementation. Third, it remains open whether a similar syntax directed decision procedure for quasi-reducibility can be designed, i.e., where matching may occur in arbitrary subterms. Finally, one might consider an extension where it is allowed to add structural axioms to some symbols such as associativity and commutativity.

References

1 Takahito Aoto and Yoshihito Toyama. Ground confluence prover based on rewriting induction. In Delia Kesner and Brigitte 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. doi:10.4230/LIPICS.FSCD.2016.33.

2 Adel Bouhoula. Simultaneous checking of completeness and ground confluence for algebraic specifications. *ACM Trans. Comput. Log.*, 10(3):20:1–20:33, 2009. doi:10.1145/1507244.1507250.

3 Adel Bouhoula and Florent Jacquemard. Sufficient completeness verification for conditional and constrained TRS. *J. Appl. Log.*, 10(1):127–143, 2012. doi:10.1016/j.jal.2011.09.001.

4 Hubert Comon. Sufficient completness, term rewriting systems and "anti-unification". In Jörg H. Siekmann, editor, *8th International Conference on Automated Deduction, Oxford, England, July 27 - August 1, 1986, Proceedings*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1986. doi:10.1007/3-540-16780-3_85.

5 Stephan Falke and Deepak Kapur. Rewriting induction + linear arithmetic = decision procedure. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 241–255. Springer, 2012. doi:10.1007/978-3-642-31365-3_20.

6 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artif. Intell.*, 301:103572, 2021. doi:10.1016/J.ARTINT.2021.103572.

7 Deepak Kapur, Paliath Narendran, Daniel J. Rosenkrantz, and Hantao Zhang. Sufficient-completeness, ground-reducibility and their complexity. *Acta Informatica*, 28(4):311–350, 1991. doi:10.1007/BF01893885.

8 Deepak Kapur, Paliath Narendran, and Hantao Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24(4):395–415, 1987. doi:10.1007/BF00292110.

9 Cynthia Kop. Quasi-reductivity of logically constrained term rewriting systems. *CoRR*, 2017. arXiv:1702.02397.

10 Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reasoning*, 44(4):303–336, 2010. doi:10.1007/s10817-009-9157-2.

11 Alain Laville. Lazy pattern matching in the ML language. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science, Seventh Conference, Pune, India, December 17-19, 1987, Proceedings*, volume 287 of *Lecture Notes in Computer Science*, pages 400–419. Springer, 1987. doi:10.1007/3-540-18625-5_64.

12 Azeddine Lazrek, Pierre Lescanne, and Jean-Jacques Thiel. Tools for proving inductive equalities, relative completeness, and omega-completeness. *Inf. Comput.*, 84(1):47–70, 1990. doi:10.1016/0890-5401(90)90033-E.

13 Aart Middeldorp, Alexander Lochmann, and Fabian Mitterwallner. First-order theory of rewriting for linear variable-separated rewrite systems: Automation, formalization, certification. *J. Autom. Reason.*, 67(2):14, 2023. doi:10.1007/S10817-023-09661-7.

14 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. `doi:10.1007/3-540-45949-9`.

15 Uday S. Reddy. Term rewriting induction. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, volume 449 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 1990. `doi:10.1007/3-540-52885-7_86`.

16 Peter Sestoft. MK pattern match compilation and partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 1996. `doi:10.1007/3-540-61580-6_22`.

17 Christian Sternagel and René Thiemann. First-order terms. *Archive of Formal Proofs*, February 2018. , Formal proof development. URL: `https://isa-afp.org/entries/First_Order_Terms.html`.

18 Jean-Jacques Thiel. Stop losing sleep over incomplete data type specifications. In Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, editors, *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 76–82. ACM Press, 1984. `doi:10.1145/800017.800518`.