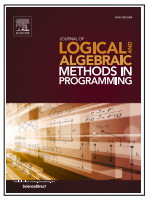
J. Log. Algebraic Methods Program. 151 (2026) 101129

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



A verified algorithm for deciding pattern completeness with optimal asymptotic complexity ⋆

René Thiemann [a,*], Akihisa Yamada [b]

[a] University of Innsbruck, Techniker Straße 21a, 6020, Innsbruck, Austria
[b] National Institute of Advanced Industrial Science and Technology, 2-3-26 Aomi, Koto-ku, 135-0064, Tokyo, Japan

ARTICLE INFO

Keywords:
Isabelle/HOL
Pattern matching
Term rewriting

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. The algorithm has an asymptotic optimal complexity, as it exhibits a co-NP behavior.

Despite the simple structure, it is not immediate to either prove the correctness of the algorithm or the co-NP behavior. 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.

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 [2] in the context of TRSs or pattern completeness in the context of functional programming. For instance in Isabelle/HOL [3], 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 [4,5] require similar completeness results, e.g., for proving ground confluence.

Example 1. Let $\mathcal{C}_{\mathbb{N}} = \{\mathsf{true} : \mathbb{B},\ \mathsf{false} : \mathbb{B},\ 0 : \mathbb{N},\ \mathsf{s} : \mathbb{N} \to \mathbb{N}\}$ 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.

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

⋆ This research was funded in part by the Austrian Science Fund (FWF) [Grant 10.55776/I5943]
* Corresponding author.
E-mail addresses: rene.thiemann@uibk.ac.at (R. Thiemann), akih.yamada@gmail.com (A. Yamada).

https://doi.org/10.1016/j.jlamp.2026.101129
Received 31 March 2025; Received in revised form 31 March 2026; Accepted 6 April 2026
Available online 22 April 2026

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 [2]. 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* [6] 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 [7].

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.
- In our experiments it outperforms existing implementations of the complement algorithm, the approach via tree automata, and the pattern completeness check by the `ghc` Haskell compiler.
- Our algorithm can be seen as a non-deterministic polynomial-time decision procedure for pattern *in*completeness. In this sense the algorithm is asymptotically optimal as the problem is in co-NP.
- Its correctness is fully verified in Isabelle/HOL.
- The stated complexity is partially verified in Isabelle/HOL: a polynomial upper bound on the number of abstract steps is part of the formalization. However, we did not formally prove that every abstract step can be executed in polynomial time.
- We also provide a restricted version of the algorithm; it only handles the linear case, is completely syntax directed, and easy to implement.

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 $x > 0$" [8,9], but both algorithms do not properly generalize the result of Kapur et al. since they are restricted to linear lhss. Bouhoula and Jacquemard [10] 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 [11], 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` [12], but as many other algorithms for pattern completeness, it is restricted to the left-linear case.

There are also algorithms to compile pattern matching [13,14], 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.

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 algorithm that covers the linear case. The algorithm is then extended to handle the general case in Section 4. The algorithms of Sections 3 and 4 have already been presented in our FSCD paper [1], a preliminary version of this article. In particular, the FSCD algorithm requires exponential space and is therefore not optimal. To this end, we modify the FSCD algorithm of Section 4 to a novel algorithm that is presented in Section 5. Afterwards we present details on the Isabelle formalization and on the implementation in Section 6. The experimental results are provided in Section 7 before we conclude in Section 8.

The formalization, the executable code and details on the experiments are available at:
https://isafor-ceta.uibk.ac.at/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 finite set of function symbols f, each associated with a nonempty sequence of sorts $\iota_1, \ldots, \iota_n, \iota_0 \in \mathcal{S}$, 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 $f(t_1, \ldots, t_n) : \iota_0 \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ if $f : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{F}$ and $t_1 : \iota_1, \ldots, t_n : \iota_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. 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

[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.

[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.

composition of two substitutions σ and δ, i.e., $t(\sigma\delta) = (t\sigma)\delta$, 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 example a program might be a TRS, or some other first-order functional programming language that uses pattern matching. We assume the signature $\mathcal{F}$ is split into two disjoint signatures $\mathcal{C}$ and $\mathcal{D}$, where $\mathcal{C}$ contains *constructor* symbols and $\mathcal{D}$ contains *defined* symbols, and values are represented by constructor ground terms in $\mathcal{T}(\mathcal{C})$. The *cardinality* $|\iota|$ of a sort $\iota \in \mathcal{S}$ is defined as the number of values of that sort, i.e., $|\{t \mid t : \iota \in \mathcal{T}(\mathcal{C})\}|$. We assume that every sort is *inhabited*, i.e., $|\iota| > 0$. A sort ι is *finite* if $|\iota|$ is finite, and is infinite otherwise. We say a term $f(t_1, \ldots, t_n)$ is *basic* if $f : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{D}$ and $t_1 : \iota_1, \ldots, t_n : \iota_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, and denote the (sorted) set of basic terms by $\mathcal{T}_B(\mathcal{C}, \mathcal{D}, \mathcal{V})$.

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 $t \in \mathcal{T}_B(\mathcal{C}, \mathcal{D}, \emptyset)$ 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 [13,15,16]: 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 property that L covers $T := \{f(x_1, \ldots, x_n) \mid f : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{D}\}$ where $x_1, \ldots, x_n$ is a sequence of distinct variables; and the complement algorithm of Lazrek et al. [6] 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 [2], 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 $t \in \mathcal{T}_B(\mathcal{C}, \mathcal{D}, \emptyset)$ 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 $\mathcal{C}_\mathbb{Z} = \{\mathsf{true} : \mathbb{B}, \mathsf{false} : \mathbb{B}, 0 : \mathbb{Z}, \mathsf{s} : \mathbb{Z} \to \mathbb{Z}, \mathsf{p} : \mathbb{Z} \to \mathbb{Z}\}$ to represent the Booleans and integers in a successor–predecessor notation, e.g., $\mathsf{p}(0)$ 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.

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

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

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 $\mathsf{s}^i(0)$ or $\mathsf{p}^i(0)$ and then rules (1) or (2) will be applicable. On the other hand, the TRS is not pattern complete since $\mathsf{even}(\mathsf{s}(\mathsf{p}(0)))$ is not matched by any lhs.

3. Pattern completeness – the linear case

Before we design the 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 atom* is a pair of terms written $t \sim \ell$, where $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is called *matchee* and $\ell \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is called *pattern*.

A *matching problem* is a finite set of matching atoms, and a *pattern problem* is a finite set 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 $t \sim \ell \in mp$. A pattern problem pp is complete if for each $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C})$ there is some $mp \in pp$ such that mp is complete w.r.t. σ.

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

pp is complete iff

$$\forall \sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C}).\ \exists mp \in pp.\ \exists \gamma : \mathcal{X} \to \mathcal{T}(\mathcal{F}).\ \forall t \sim \ell \in mp.\ t\sigma = \ell\gamma$$

Pattern problems are quite generic and can express several properties. For instance, L covers T iff all pattern problems in $P = \{\{\{t \sim \ell\} \mid \ell \in L\} \mid t \in T\}$ are complete. Similarly, Aoto and Toyama's notion of strong quasi-reducibility [4] can also be encoded as a pattern problem: informally, this notion expresses that any constructor ground instance of $f(x_1, \ldots, x_n)$ has a redex w.r.t. lhss L at the root or directly below the root. This property is equivalently expressed as completeness of the pattern problem $\{\{t \sim \ell\} \mid t \in \{f(x_1, \ldots, x_n), x_1, \ldots, x_n\},\ \ell \in L\}$. 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 all pattern problems within $P = \{\{\{f(x_1, \ldots, x_n) \sim \ell\} \mid \ell \in L\} \mid f : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{D}\}$.

The following inference rules describe simplification rules for arbitrary pattern problems, and these rules form a decision procedure to determine the completeness of *linear* pattern problems. We say a matching problem $\{t_1 \sim \ell_1, \ldots, t_n \sim \ell_n\}$ is *linear* if each ℓ_i is linear

and $\mathsf{Var}(\ell_i) \cap \mathsf{Var}(\ell_j) = \emptyset$ for $i \neq j$, and say a pattern problem pp is linear if all $mp \in pp$ are linear. Note that these inference rules are equivalent to the ones in the FSCD paper [1], but they are not identical. The main difference is the kind of non-determinism. In the FSCD paper only one kind of non-determinism was present. In contrast, in the upcoming presentation we introduce two forms of non-determinism, which will be essential for the complexity analysis.

To be more precise, the first kind of non-determinism—which is also present in the FSCD paper—is that there can be multiple rules that are applicable. This non-determinism is of the "don't care" type in the sense, that it does not matter in which order the rules are applied. Here, any concrete implementation can choose a strategy on its own on which rules are tried to be applied first, and once an applicable rule on some problem pp has been detected, there is no demand to apply any other rule on pp.

Second, there is also non-determinism that enables a co-NP behavior: The rules transform a single pattern problem pp into a set of new pattern problems $pp_1, \ldots, pp_n$, and for these rules there is the connection that pp is complete iff all of the pp_i's are complete. Hence, the non-deterministic algorithm will be able to select some incomplete pp_i and continue with that one, in case any of the problems $pp_1, \ldots, pp_n$ is incomplete. On the other hand, if pp is pattern complete, then one needs to exhaustively consider all new pattern problems $pp_1, \ldots, pp_n$ and check their completeness property.

To avoid confusion, we denote the empty matching problem by $\top_{mp}$ (nothing has to be matched), the empty pattern problem by $\bot_{pp}$ (no candidate pattern can match), and the empty set of pattern problems by $\top_P$ (no problem to be solved). By definition, $\top_{mp}$ is complete for any ground substitution, and $\bot_{pp}$ is incomplete. In the inference rules we also extend the set of matching problems by a special matching problem $\bot_{mp}$ that represents a matching problem which is not complete w.r.t. any constructor ground substitution. Similarly, we define $\top_{pp}$ as a new pattern problem that is always complete.

Definition 6 (Reduction Rules for Linear Pattern Problems)**.** We define the reduction relation $\to$ from matching problems to extended matching problems by the following rules.

$$\{f(t_1, \ldots, t_n) \sim f(\ell_1, \ldots, \ell_n)\} \uplus mp \to \{t_1 \sim \ell_1, \ldots, t_n \sim \ell_n\} \cup mp \qquad \text{(decompose)}$$

$$\{t \sim x\} \uplus mp \to mp \qquad \text{if } \forall s \sim \ell \in mp.\ x \notin \mathsf{Var}(\ell) \qquad \text{(match)}$$

$$\{f(\ldots) \sim g(\ldots)\} \uplus mp \to \bot_{mp} \qquad \text{if } f \neq g \qquad \text{(clash)}$$

On top of this we define the relation $\Rightarrow$ from pattern problems to sets of pattern problems as follows:

$$\{mp\} \uplus pp \Rightarrow \{\{mp'\} \cup pp\} \qquad \text{if } mp \to mp' \neq \bot_{mp} \qquad \text{(simp-mp)}$$

$$\{mp\} \uplus pp \Rightarrow \{pp\} \qquad \text{if } mp \to \bot_{mp} \qquad \text{(remove-mp)}$$

$$\{\top_{mp}\} \uplus pp \Rightarrow \top_P \qquad \text{(success)}$$

$$pp \Rightarrow \mathsf{Inst}(pp, x) \qquad \text{if } x \sim f(\ldots) \in mp \in pp \qquad \text{(instantiate)}$$

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 $pp\sigma_{x,c} = \{\{t\sigma_{x,c} \sim \ell \mid t \sim \ell \in mp\} \mid mp \in pp\}$ for each $c : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{C}$, where $\sigma_{x,c} = [x \mapsto c(x_1, \ldots, x_n)]$ for distinct fresh variables $x_1 : \iota_1, \ldots, x_n : \iota_n \in \mathcal{V}$.

As a special case of (match), one has $\{t \sim x\} \to \emptyset = \top_{mp}$; similarly, as a special case of (remove-mp), one has $\{mp\} \Rightarrow \{\emptyset\} = \{\bot_{pp}\}$ when $mp \to \bot_{mp}$.

We further define the reduction relation $\Rightarrow_{nd}$ from pattern problems to extended pattern problems , which makes non-deterministic choices: $pp \Rightarrow_{nd} pp'$ iff $pp \Rightarrow P$ and $pp' \in P$, and $pp \Rightarrow_{nd} \top_{pp}$ iff $pp \Rightarrow \top_P$.

Clearly, rules (decompose), (match) and (clash) correspond to a standard matching algorithm, and (simp-mp) and (remove-mp) lift these simplifications on matching problems to pattern problems. The (success) rule identifies solved matching problems in which case the whole pattern problem is pattern complete, so no new problems are generated. Finally there is the (instantiate) rule. Here a matching algorithm would detect a failure since a variable x is never matched by a non-variable term $f(\ldots)$. 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 $c(x_1, \ldots, x_n)$ for all $c : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in \mathcal{C}$.

For a binary relation $\to$, $\to^*$ denotes its reflexive transitive closure, and $\to^!$ is defined as reduction to normal form, i.e., $x \to^! y$ iff $x \to^* y$ and there exist no z with $y \Rightarrow_{nd} z$. The following theorem gives an algorithm for deciding pattern completeness of linear pattern problems.

Theorem 7 (Decision Procedure for Linear Pattern Problems)**.** *A linear pattern problem pp is complete iff $pp \Rightarrow_{nd}^! \bot_{pp}$ does not hold, which is decidable. More precisely:*

- *$\Rightarrow_{nd}$ is terminating.*
- *If $pp \Rightarrow P'$, then pp is complete iff all $pp' \in P'$ are complete.*
- *If pp is linear and $pp \Rightarrow_{nd} pp'$, then pp' is linear.*
- *If pp is linear and $pp \Rightarrow_{nd}^! pp'$, then $pp' \in \{\top_{pp}, \bot_{pp}\}$.*
- *If pp is linear, then pp is incomplete iff $pp \Rightarrow_{nd}^! \bot_{pp}$.*

Proof. The property that normal forms will be either $\top_{pp}$ or $\bot_{pp}$ follows by an easy analysis of the rules. In particular (clash), (decompose), (match), and (instantiate) cover all cases of $t \sim \ell$ 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 argued as follows. First, it is easy to check that if $mp \to mp'$, then for every σ, mp is complete with respect to σ iff so is mp'. In combination with incompleteness of $\bot_{mp}$ and completeness of $\top_{mp}$, this shows that rules (simp-mp), (remove-mp), and (success) preserve completeness. It remains to treat rule (instantiate), but its completeness was already explained in the paragraph directly after Definition 6.

Preservation of linearity follows by an easy analysis of the rules: the only interesting rule is (instantiate) where new matchee terms are introduced by applying $\sigma_{x,c} = [x \mapsto c(x_1, \dots, x_n)]$; and this substitution preserves linearity, since the variables $x_1, \dots, x_n$ are distinct and fresh.

The most interesting aspect is termination. To prove it, we first define a measure $|t \sim \ell|_{\mathsf{diff}}$ for a matching atom $t \sim \ell$:

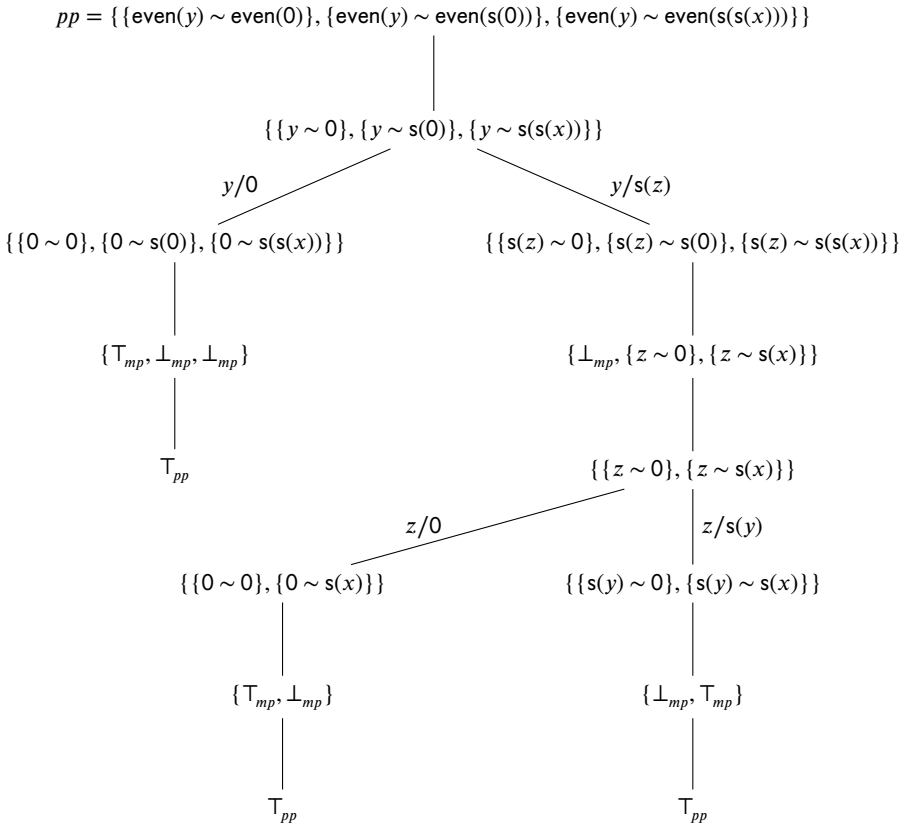
- $|x \sim \ell|_{\mathsf{diff}}$ is the number of function symbols in ℓ,
- $|f(t_1, \dots, t_n) \sim f(\ell_1, \dots, \ell_n)|_{\mathsf{diff}} = \sum_{i=1}^{n} |t_i \sim \ell_i|_{\mathsf{diff}}$, and
- $|t \sim \ell|_{\mathsf{diff}} = 0$ in all other cases.

We lift this measure to pattern problems by $|pp|_{\mathsf{diff}} = \sum_{t \sim \ell \in mp \in pp} |t \sim \ell|_{\mathsf{diff}}$. This measure has the nice property that each $\Rightarrow_{nd}$-step weakly decreases and any $\Rightarrow_{nd}$-step originating from the (instantiate) rule strictly decreases w.r.t. this measure. Hence, (instantiate) cannot be applied infinitely often. That the remaining rules terminate does not need a complicated measure: their application decreases the number of symbols of matchee terms t in a pattern problem.

Finally, we can assemble all previous results to prove the theorem. For the one direction we assume $pp \Rightarrow_{nd}^{!} \bot_{pp}$. Then in particular $pp \Rightarrow_{nd}^{*} \bot_{pp}$ and by induction on the length of this derivation we obtain that pp must be incomplete, using preservation of completeness. For the other direction we assume that pp is linear and incomplete. Now we perform induction on $\Rightarrow_{nd}$ to prove $pp \Rightarrow_{nd}^{!} \bot_{pp}$. If pp is already in normal form w.r.t. $\Rightarrow_{nd}$, then by the shape of normal forms we know that pp must be $\top_{pp}$ or $\bot_{pp}$. Using that pp is incomplete, we conclude $pp = \bot_{pp}$ and hence, $pp \Rightarrow_{nd}^{!} \bot_{pp}$. In the other case, pp is not in normal form. Hence, there must be some pp' such that $pp \Rightarrow_{nd} pp'$, and by the definition of $\Rightarrow_{nd}$ there is some P satisfying $pp' \in P$ and $pp \Rightarrow P$. Using the completeness of $\Rightarrow$ in combination with incompleteness of pp, there is at least one incomplete pattern problem $pp'' \in P$. By preservation of linearity and the induction hypothesis, we conclude $pp'' \Rightarrow_{nd}^{!} \bot_{pp}$ and hence $pp \Rightarrow_{nd} pp'' \Rightarrow_{nd}^{!} \bot_{pp}$. □

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. A branching in the tree indicates an application of the (instantiate)-rule, and each other edge in the tree represents a sequence of non-branching $\Rightarrow_{nd}$ steps.



Regarding the complexity of the algorithm for linear pattern problems, we will prove that our algorithm is optimal. Note that deciding quasi-reducibility is co-NP complete [17], and this result carries over to pattern completeness: the restriction to just search for matches at the root position in pattern completeness does not lead to a lower complexity class.

We show that pattern completeness is co-NP hard, and afterwards show that a non-deterministic run of our algorithm returns in polynomial time for incomplete pattern problems.

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 $\alpha \vDash c_i$ 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 $pp_\varphi := \{\{f(x_1, \ldots, x_n) \sim \ell_1\}, \ldots, \{f(x_1, \ldots, x_n) \sim \ell_m\}\}$ and derive that φ is unsatisfiable iff pp_φ is complete. □

In order to show the non-deterministic polynomial time complexity of the algorithm for linear pattern problems, we will adjust the measure that was used for the termination proof in a way that we get a strict decrease for all rules, and that the measure is polynomially bounded in the size of the pattern problem. To this end, let us be more precise on the currently utilized measures and the results so far.

Definition 10 (Symbols). We define $\mathsf{syms}(t)$ as the multiset of symbols in a term t.

- $\mathsf{syms}(x) = \{x\}$ if $x \in \mathcal{X}$, and
- $\mathsf{syms}(f(t_1, \ldots, t_n)) = \{f\} \cup \mathsf{syms}(t_1) \cup \ldots \cup \mathsf{syms}(t_n)$

For a pattern problem, we further define the multiset of symbols of matchee terms t within the pattern problem.

$$\mathsf{tsyms}(pp) := \bigcup_{t \sim \ell \in mp \in pp} \mathsf{syms}(t)$$

The size of matchee terms within a pattern problem is defined as

$$|pp|_{\mathsf{tsize}} := |\mathsf{tsyms}(pp)|$$

Lemma 11 (Size Estimations). *Assume that $pp \Rightarrow P$ and $pp' \in P$.*

- *If $pp \Rightarrow P$ is due to rules (simp-mp) or (remove-mp), then $\mathsf{tsyms}(pp) \supset \mathsf{tsyms}(pp')$ and hence $|pp|_{\mathsf{tsize}} > |pp'|_{\mathsf{tsize}}$. Moreover, $|pp|_{\mathsf{diff}} \geq |pp'|_{\mathsf{diff}}$.*
- *If $pp \Rightarrow P$ is due to rule (instantiate), then $|pp|_{\mathsf{diff}} > |pp'|_{\mathsf{diff}}$, but $|pp|_{\mathsf{tsize}} \leq |pp'|_{\mathsf{tsize}}$.*

The challenge at this point is to develop a combined measure such that the increase of $|\cdot|_{\mathsf{tsize}}$ in the (instantiate) case is compensated by the decrease of $|\cdot|_{\mathsf{diff}}$. To this end, let us briefly recall to what extend $|\cdot|_{\mathsf{tsize}}$ is increased via (instantiate).

During the (instantiate) step a variable x is replaced by $c(x_1, \ldots, x_n)$. Each such replacement increases the number of symbols by exactly n, which can be bounded from above by the maximum arity of the symbols in $\mathcal{C}$. In order to know how often this replacement happens, we have to count how often the variable x occurs in a pattern problem. We will bound this number from above by yet another measure, $|\cdot|_{\mathsf{dup}}$, that expresses the maximum duplication factor of variables in matchee terms. With this auxiliary measure we will then design the overall measure $|\cdot|_{\alpha,\mathsf{pat}}$ for pattern problems.

Definition 12 (Measure for Pattern Problems). The *maximum duplication factor* of a pattern problem is defined as

$$|pp|_{\mathsf{dup}} := \max\{\mathsf{count}(x, \mathsf{tsyms}(pp)) \mid x \in \mathcal{X}\}$$

where $\mathsf{count}(x, M)$ counts how often x occurs in multiset M.

The overall measure for pattern problems is parametrized by some constant $\alpha \in \mathbb{N}$.[3] It is defined as

$$|pp|_{\alpha,\mathsf{pat}} := (\alpha + |pp|_{\mathsf{diff}}) \cdot (|pp|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp|_{\mathsf{tsize}}$$

where m is the maximum arity of the symbols in $\mathcal{C}$. The input size of a pattern problem is defined as

$$|pp|_{\mathsf{size}} := \sum_{t \sim \ell \in mp \in pp} (|t| + |\ell|)$$

[3] For the upcoming Theorem 15 we do not really need α and can set it to 0, but α will be required later for the proof of Theorem 40.

It is easy to see that $|pp|_{\alpha,\mathsf{pat}}$ is polynomially bounded from above by the input size $|pp|_{\mathsf{size}}$, since $\max\{|pp|_{\mathsf{diff}}, |pp|_{\mathsf{dup}}, |pp|_{\mathsf{tsize}}\} \leq |pp|_{\mathsf{size}}$.

Lemma 13 (Connecting $|pp|_{\alpha,\mathsf{pat}}$ and $|pp|_{\mathsf{size}}$)**.** $|pp|_{\alpha,\mathsf{pat}} \leq (\alpha + |pp|_{\mathsf{size}}) \cdot (|pp|_{\mathsf{size}} \cdot \mathsf{m} + 2)$.

Lemma 14 (Extended Size Estimation)**.** *If $pp \Rightarrow P$ and $pp' \in P$, then $|pp|_{\alpha,\mathsf{pat}} > |pp'|_{\alpha,\mathsf{pat}}$.*

Proof. If $pp \Rightarrow P$ was caused by rules (simp-mp) or (remove-mp), then $\mathsf{tsyms}(pp) \supset \mathsf{tsyms}(pp')$ and hence $|pp|_{\mathsf{dup}} \geq |pp'|_{\mathsf{dup}}$. In combination with Lemma 11, we conclude $|pp|_{\alpha,\mathsf{pat}} > |pp'|_{\alpha,\mathsf{pat}}$.

Suppose that $pp \Rightarrow P$ was due to rule (instantiate). Then we have $|pp|_{\mathsf{tsize}} + |pp|_{\mathsf{dup}} \cdot \mathsf{m} \geq |pp'|_{\mathsf{tsize}}$ from the discussion above. We also have $|pp|_{\mathsf{dup}} \geq |pp'|_{\mathsf{dup}}$ since $\mathsf{count}(x_i, \mathsf{tsyms}(pp')) = \mathsf{count}(x, \mathsf{tsyms}(pp))$ for each fresh variable x_i introduced when substituting x by $c(x_1, \ldots, x_n)$.

The desired decrease is obtained as follows.

$$\begin{aligned}
|pp|_{\alpha,\mathsf{pat}} &= (\alpha + |pp|_{\mathsf{diff}}) \cdot (|pp|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp|_{\mathsf{tsize}} \\
&\geq (\alpha + |pp'|_{\mathsf{diff}} + 1) \cdot (|pp|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp|_{\mathsf{tsize}} \\
&= (\alpha + |pp'|_{\mathsf{diff}}) \cdot (|pp|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp|_{\mathsf{tsize}} + |pp|_{\mathsf{dup}} \cdot \mathsf{m} + 1 \\
&> (\alpha + |pp'|_{\mathsf{diff}}) \cdot (|pp|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp|_{\mathsf{tsize}} + |pp|_{\mathsf{dup}} \cdot \mathsf{m} \\
&\geq (\alpha + |pp'|_{\mathsf{diff}}) \cdot (|pp|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp'|_{\mathsf{tsize}} \\
&\geq (\alpha + |pp'|_{\mathsf{diff}}) \cdot (|pp'|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp'|_{\mathsf{tsize}} \\
&= |pp'|_{\alpha,\mathsf{pat}} \qquad \square
\end{aligned}$$

Theorem 15 (Linear Pattern Problems in co-NP)**.** *Completeness of linear pattern problems is in co-NP, i.e., incompleteness is NP.*

Proof. Our algorithm is to non-deterministically reduce the input by $\Rightarrow_{nd}$ and return false when reaching $\bot_{pp}$. This algorithm has non-deterministic polynomial-time complexity, since:

- If $pp \Rightarrow_{nd} pp'$, then $|pp|_{\alpha,\mathsf{pat}} > |pp'|_{\alpha,\mathsf{pat}}$ by Lemma 14.
- If $pp \Rightarrow_{nd}^{n} pp'$, then $|pp|_{\alpha,\mathsf{pat}} \geq |pp'|_{\alpha,\mathsf{pat}} + n$.
- The maximum length of sequences of $\Rightarrow_{nd}$-steps starting from pp is bounded from above by $|pp|_{0,\mathsf{pat}}$ and hence also by $|pp|_{\mathsf{size}} \cdot (|pp|_{\mathsf{size}} \cdot \mathsf{m} + 2)$ by using the definition of $|pp|_{0,\mathsf{pat}}$ in Definition 12. In total, this given a polynomial upper bound w.r.t. the input size.
- Each $\Rightarrow_{nd}$-step can be implemented in polynomial time. $\square$

In the remainder of this section we study two variants of the non-deterministic algorithm. We illustrate how to turn the non-deterministic algorithm that uses both kinds of non-determinism into an algorithm that only includes don't-care non-determinism, and we show how to integrate counterexample generation.

In order to get rid of the second kind of non-determinism, instead of working on single pattern problem, one now stores a full set of pattern problems, namely, those pattern problems in the search tree that still need to be investigated. As a consequence, the only non-determinism that remains is the selection of rules, and this selection is not important for the result. If one further fixes some order on the application of rules and also fixes some selection strategy for matching atoms and matching problems and pattern problems, then one obtains a fully deterministic algorithm.

Definition 16. We define $\Rrightarrow$ as a relation between sets of pattern problems and extended sets of pattern problems, where $\bot_P$ is a special set of pattern problems that encodes incompleteness.

$$\{pp\} \uplus P \Rrightarrow P' \cup P \quad \text{if } pp \Rightarrow P' \tag{simp-pp}$$

$$\{\bot_{pp}\} \uplus P \Rrightarrow \bot_P \tag{failure}$$

Note that $\Rrightarrow$ has already been presented in the FSCD paper, but there everything has been proven directly for $\Rrightarrow$ and there has not been an NP algorithm for deciding pattern incompleteness.

The soundness of $\Rrightarrow$ is an easy consequence of Theorem 7, where the main difference is that instead of considering *all* sequences of $\Rightarrow$ steps, one just has to consider a *single* sequence of $\Rrightarrow$ steps.

Theorem 17 (Don't Care Non-Deterministic Decision Procedure for Completeness of Linear Pattern Problems)**.**

- *$\Rrightarrow$ is terminating.*
- *If $P \Rrightarrow P'$, then all $pp \in P$ are complete iff all $pp' \in P'$ are complete.*
- *If all $pp \in P$ are linear and $P \Rrightarrow P'$, then all $pp' \in P'$ are linear.*
- *If all $pp \in P$ are linear and $P \Rrightarrow^{!} P'$, then $P' \in \{\top_P, \bot_P\}$.*
- *If pp is linear and $\{pp\} \Rrightarrow^{!} P$, then pp is complete iff $P = \top_P$.*

Example 18. The computation in Example 8 is simulated in the don't-care non-deterministic algorithm as follows, where the singleton steps are exactly the applications of the (instantiate) rule.

$$
\begin{aligned}
&\{\,\{\{\mathsf{even}(y) \sim \mathsf{even}(0)\}, \{\mathsf{even}(y) \sim \mathsf{even}(\mathsf{s}(0))\}, \{\mathsf{even}(y) \sim \mathsf{even}(\mathsf{s}(\mathsf{s}(x)))\}\}\,\}\\
&\Rrightarrow^* \ \{\,\{\{y \sim 0\}, \{y \sim \mathsf{s}(0)\}, \{y \sim \mathsf{s}(\mathsf{s}(x))\}\}\,\}\\
&\Rrightarrow \ \{\,\{\{0 \sim 0\}, \{0 \sim \mathsf{s}(0)\}, \{0 \sim \mathsf{s}(\mathsf{s}(x))\}\},\\
&\qquad \{\{\mathsf{s}(z) \sim 0\}, \{\mathsf{s}(z) \sim \mathsf{s}(0)\}, \{\mathsf{s}(z) \sim \mathsf{s}(\mathsf{s}(x))\}\}\,\}\\
&\Rrightarrow^* \ \{\,\{\{\top_{mp}, \bot_{mp}, \bot_{mp}\},\\
&\qquad \{\bot_{mp}, \{z \sim 0\}, \{z \sim \mathsf{s}(x)\}\}\,\}\\
&\Rrightarrow^* \ \{\,\{\{z \sim 0\}, \{z \sim \mathsf{s}(x)\}\}\,\}\\
&\Rrightarrow \ \{\,\{\{0 \sim 0\}, \{0 \sim \mathsf{s}(x)\}\}, \ \{\{\mathsf{s}(y) \sim 0\}, \{\mathsf{s}(y) \sim \mathsf{s}(x)\}\}\,\}\\
&\Rrightarrow^* \ \{\,\{\top_{mp}, \bot_{mp}\}, \ \{\bot_{mp}, \top_{mp}\}\,\}\\
&\Rrightarrow^* \ \top_P
\end{aligned}
$$

If one implements a strategy of $\Rrightarrow$ that corresponds to a depth-first search of the non-deterministic computation, then one obtains an algorithm which runs with polynomial space usage and has at most a single exponential time consumption, since a full computation tree of $\Rightarrow_{nd}$ has polynomial depth and a branching factor of at most $|C|$.

Note that both $\Rightarrow_{nd}$ and $\Rrightarrow$ can be modified to support counterexample generation, i.e., in case of a negative result one returns a constructor substitution that violates the completeness of the initial pattern problem. To achieve this, for each pattern problem we additionally store a constructor substitution δ during the execution of the algorithms. This substitution is initially empty, i.e., nothing is instantiated, and δ will keep track of the instantiations that are performed during the algorithm. For instance, the inference rules of $\Rrightarrow$ are modified as follows:

$$
\begin{aligned}
(\{mp\} \uplus pp, \delta) &\Rrightarrow \{(\{\{mp'\} \cup pp\}, \delta)\} && \text{if } mp \to mp' \neq \bot_{mp} && \text{(simp-mp)}\\
(\{mp\} \uplus pp, \delta) &\Rrightarrow \{(\{pp\}, \delta)\} && \text{if } mp \to \bot_{mp} && \text{(remove-mp)}\\
(\{\top_{mp}\} \uplus pp, \delta) &\Rrightarrow \top_P && && \text{(success)}\\
(pp, \delta) &\Rrightarrow \{(pp\sigma_{x,c}, \delta\sigma_{x,c}) \mid c : \cdots \in C\} && \text{if } \ldots && \text{(instantiate)}
\end{aligned}
$$

Here, the dots refer to the original definitions and conditions of the (instantiate) rule where the definition of $\mathsf{Inst}(pp, x)$ has been expanded.

4. Pattern completeness – the general case

In the non-linear case both $\Rightarrow_{nd}$ and $\Rrightarrow$—might get stuck, e.g., if there is a matching problem $\{t \sim x, t' \sim x\}$ for $t \neq t'$, one obtains normal forms which are different from $\bot_{pp}$, $\top_{pp}$, $\bot_P$ and $\top_P$. To treat such cases we have to add further simplification rules. In order to do so without breaking termination, we need to take finiteness of sorts into account, cf. Example 19.

Example 19. Consider a TRS to determine the majority of three values: $\{\mathsf{f}(x,x,y) \to x, \mathsf{f}(x,y,x) \to x, \mathsf{f}(y,x,x) \to x\}$. 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 contains at least three different ground terms a, b, c, then no lhs matches $\mathsf{f}(a,b,c)$. In order to figure out which of these cases applies, we may want to instantiate the variables in $\mathsf{f}(x_1,x_2,x_3)$, so that the algorithm can proceed.

So, we want to allow instantiating variables to judge pattern completeness. For reasons that will be explained later, we only allow instantiating variables of finite sorts.

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

Definition 20. 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. In this case x is named a conflict variable, and the other term in $\{t|_p, t'|_p\}$ is named a conflict term.

We remark that there is a different flavor of problems with non-linear matching problems of the form $\{t \sim y, t' \sim y\}$. A clash of t and t' immediately leads to matching failure. If there is a difference in variable x of finite sort, this can also be handled locally by instantiating all possible options of x. On the other hand, when x is of infinite sort, one needs a special care.

Definition 21. Two terms t and t' have an *infinite variable difference* if t and t' differ in variable $x : \iota \in \mathcal{V}$ and ι is infinite. A matching problem mp has an infinite variable difference, if there are t, t' and y such that $\{t \sim y, t' \sim y\} \subseteq mp$, and t and t' have an infinite variable difference.

We remark that it is easy to compute whether a matching problem contains an infinite variable difference, assuming that for every sort ι we know whether it is infinite or finite. In our implementation, we precompute this sort information from the sort descriptions, see page 21 for more details.

By adding three rules for the three kinds of differences described above, we obtain the inference rules for non-linear pattern problems.

Definition 22 (Reduction Rules for Arbitrary Pattern Problems). We take all rules of Definition 6 and add the following ones.

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

$$pp \Rightarrow \{\bot_{pp}\} \quad \text{if each } mp \in pp \text{ has an infinite variable difference and } pp \neq \bot_{pp} \qquad \text{(inf-diff)}$$

$$pp \Rightarrow \mathsf{Inst}(pp, x) \quad \text{if } t \text{ and } t' \text{ differ in variable } x : \iota \in \mathcal{V},\ \iota \text{ is finite, and } \{t \sim y, t' \sim y\} \subseteq mp \in pp \qquad \text{(instantiate')}$$

The relations $\Rightarrow_{nd}$ and $\Rrightarrow$ are defined as before, but now based on the extended inference rules of $\Rightarrow$.

Indeed, with these new rules, $\Rightarrow_{nd}$ and $\Rrightarrow$ cannot get stuck even for non-linear inputs, cf. the upcoming Theorem 24.

Infinite variable differences can only be handled via (inf-diff) if all matching problems show such a difference. It is unsound to turn (inf-diff) into a local rule for matching problems, i.e., if we would make (inf-diff) similar to (clash').

Example 23. Consider lhss $\{\mathsf{f}(x,x), \mathsf{f}(\mathsf{s}(x),y), \mathsf{f}(x,\mathsf{s}(y))\}$, where $\mathsf{f} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ 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:

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

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

The reason why (instantiate') is restricted to finite sorts is to ensure termination. For instance, assuming that all variables are of sort $\mathbb{N}$, we would get an infinite sequence without the restriction:

$$\begin{aligned} &\{\{x \sim z, y \sim z\}\} \Rightarrow_{nd} \{\{\mathsf{s}(x_1) \sim z, y \sim z\}\} \\ \Rightarrow_{nd} &\{\{\mathsf{s}(x_1) \sim z, \mathsf{s}(y_1) \sim z\}\} \Rightarrow_{nd} \{\{\mathsf{s}(\mathsf{s}(x_2)) \sim z, \mathsf{s}(y_1) \sim z\}\} \\ \Rightarrow_{nd} &\{\{\mathsf{s}(\mathsf{s}(x_2)) \sim z, \mathsf{s}(\mathsf{s}(y_2)) \sim z\}\} \Rightarrow_{nd} \dots \end{aligned}$$

We state a result similarly to Theorem 7 for the linear case, though its proof is much more involved.

Theorem 24 (Non-Deterministic Decision Procedure for Completeness of Pattern Problems). *Consider $\Rightarrow_{nd}$ of Definition 22.*

- *$\Rightarrow_{nd}$ is terminating.*
- *If $pp \Rrightarrow P'$, then pp is complete iff all $pp' \in P'$ are complete.*
- *If pp and $pp \Rightarrow_{nd}^{!} pp'$, then $pp' \in \{\top_{pp}, \bot_{pp}\}$.*
- *Pattern problem pp is incomplete iff $pp \Rightarrow_{nd}^{!} \bot_{pp}$.*

Proof. It is easy to see that the only possible normal forms are $\top_{pp}$ and $\bot_{pp}$.

Termination is more complicated. We reuse the measures $|\cdot|_{\mathsf{diff}}$ and $|\cdot|_{\mathsf{tsize}}$ from the linear case, and combine it with one further measure:

- $|pp|_{\mathsf{fv\text{-}size}}$ is used to measure the size of variables of finite sort. To be more precise, for some finite sort ι, we first define $|\iota|_{\mathsf{max\text{-}size}}$ as the maximum term size of a constructor ground term of sort ι. On top of this auxiliary measure we define

$$|pp|_{\mathsf{fv\text{-}size}} = \sum_{t \sim \ell \in mp \in pp,\ x:\iota \in \mathsf{Var}(t),\ \iota \text{ is a finite sort}} |\iota|_{\mathsf{max\text{-}size}}$$

  In other words, for $|\cdot|_{\mathsf{fv\text{-}size}}$ we collect all variables of matchee terms of a pattern problem, and for each such variable x of a finite sort ι we add the maximum term size of ι.

The order $\succ$ on pattern problems is then defined as the lexicographic comparison of the triples $(|pp|_{\mathsf{diff}}, |pp|_{\mathsf{fv\text{-}size}}, |pp|_{\mathsf{tsize}})$ that are obtained from a pattern problem pp.

All rules on pattern problems give rise to a decrease w.r.t. $\succ$, as it is indicated in the following matrix: To be be more precise, consider a step $pp \Rrightarrow P$ and $pp' \in P$. Then the decreases of pp and pp' w.r.t. the utilized measures are as follows.

	$\lvert\cdot\rvert_{\mathsf{diff}}$	$\lvert\cdot\rvert_{\mathsf{fv\text{-}size}}$	$\lvert\cdot\rvert_{\mathsf{tsize}}$
(instantiate)	$>$	–	–
(instantiate')	$\geq$	$>$	–
(simp-pp), (remove-mp), (inf-diff)	$\geq$	$\geq$	$>$

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 each x_i must also be of some finite sort ι_i, and moreover $|\iota|_{\mathsf{max\text{-}size}} \geq 1 + |\iota_1|_{\mathsf{max\text{-}size}} + \dots + |\iota_n|_{\mathsf{max\text{-}size}} > |\iota_1|_{\mathsf{max\text{-}size}} + \dots + |\iota_n|_{\mathsf{max\text{-}size}}$, and hence the $|\cdot|_{\mathsf{fv\text{-}size}}$-measure strictly decreases.

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

In detail: We define that two terms t and t' have an *infinite conflict* if they clash or differ in a variable of infinite sort, and they contain no variables of finite sort. A matching problem mp has an *infinite conflict* if there are y, t, t' such that $\{t \sim y, t' \sim y\} \subseteq mp$ and t and t' have an infinite conflict. A pattern problem pp has an infinite conflict if each matching problem $mp \in pp$ has one, and we abbreviate this property by $IC(pp)$.

We will now instantiate variables in pp in a stepwise manner, maintaining IC as an invariant. Initially the infinite conflicts will be caused by differences in variables, but eventually we arrive at clashes only, which ultimately proves incompleteness.

The first step is to show that an instance of pp satisfies IC whenever (inf-diff) is applicable on pp. To this end, we define the partial ground substitution σ_0 on pp, where all variables of finite sort are instantiated by some constructor ground term of that sort. Since (inf-diff) 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. Consequently, $IC(pp\sigma_0)$ is satisfied.

Having that $pp\sigma_0$ satisfies the invariant, we now iteratively remove all infinite conflicts that are not clashes by further instantiations, $\sigma_1, \sigma_2, \ldots, \sigma_n$. To be more precise, whenever a pattern problem $pp\sigma_i$ contains an infinite conflict since t and t' differ in a variable $x : \iota \in \mathcal{V}$ in $pp\sigma_i$, we build a constructor ground term $u : \iota \in \mathcal{T}(C)$ that is larger than any of the terms in $pp\sigma_i$. Note that such u exists, since ι is infinite. We then choose $\sigma_{i+1} := \sigma_i[x \mapsto u]$ as the next instantiation. 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 such that the invariant $IC(pp\sigma_n)$ is satisfied and there are no differences of variables of infinite sort anymore. Hence, all matching problems of the final pattern problem $pp\sigma_n$ contain an infinite conflict in the form of a clash, i.e., we can extend σ_n to any constructor ground substitution σ that instantiates the remaining variables and preserves these clashes. Thus, $pp\sigma$ is incomplete by definition.

Finally we prove that the step from $pp\sigma_i$ to $pp\sigma_{i+1}$ indeed preserves IC. To this end, pick any terms t and t' of matching problem $mp \in pp\sigma_i$ that cause the infinite conflict. We show by case analysis that also $t[x \mapsto u]$ and $t'[x \mapsto u]$ have an infinite 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 whereas $t'|_p[x \mapsto u]$ contains variable y. Consequently, the latter terms have a clash along the path to variable y, or there must be a difference 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 counterexample generation is possible as in the linear case: for the (instantiate') rule one just stores σ_{xc} in the same way as it was shown for the (instantiate) rule at the end of Section 3; and the modified (inf-diff) rule with counterexample generation looks like

$$(pp, \delta) \Rightarrow \{(\bot_{pp}, \delta\sigma)\} \quad \text{if } \ldots \tag{inf-diff}$$

where the dots refer to the conditions of the original rule, and the substitution σ is defined as the constructor ground substitution that is described in the partial correctness proof of the (inf-diff) rule.

Example 25. Consider pattern problem $\{\{\mathsf{f}(z, \mathsf{s}(y)) \sim \mathsf{f}(x, x)\}\}$ with variables of sort $\mathbb{N}$ as in Example 23. A run with counterexample generation is as follows

$$(\{\{\mathsf{f}(z, \mathsf{s}(y)) \sim \mathsf{f}(x, x)\}\}, [\,]) \Rightarrow_{nd} \underbrace{(\{\{z \sim x, \mathsf{s}(y) \sim x\}\}, [\,])}_{=: pp} \Rightarrow_{nd} (\bot_{pp}, \sigma)$$

where $\sigma = [z \mapsto \mathsf{s}(\mathsf{s}(0)), y \mapsto \mathsf{s}(\mathsf{s}(\mathsf{s}(0)))]$.

The last step is due to (inf-diff) rule. Since there is no variable of finite sort, the construction of σ starts with $\sigma_0 := [\,]$. Since there is an infinite variable difference of z in pp, σ_1 instantiates z by some constructor ground term that is larger than any term in pp, e.g., $\mathsf{s}(\mathsf{s}(0))$. The resulting problem $pp[z \mapsto \mathsf{s}(\mathsf{s}(0))]$ is $\{\{\mathsf{s}(\mathsf{s}(0)) \sim x, \mathsf{s}(y) \sim x\}\}$. This problem has an infinite variable difference in y, and hence σ_2 further instantiates y with a constructor ground term that is larger than any term in $\{\{\mathsf{s}(\mathsf{s}(0)) \sim x, \mathsf{s}(y) \sim x\}\}$, e.g., $\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))$. At this point no further variables need to be instantiated and $\sigma = \sigma_2$ is the desired counterexample.

Note that the counterexample generation is not included in our verified implementation.

Whereas the soundness statements of the linear and the non-linear algorithm look very similar—compare Theorems 7 and 24—this is not the case if one considers the complexity results: there is no analogous result to Theorem 15 in the general case.

The reason for this omission is actually threefold.

- In the algorithm for the general case, a lexicographic measure is used to prove termination. It is well known, that a lexicographic combination of polynomially bounded measures is not necessarily polynomially bound itself.
- The new measure $|\cdot|_{\mathsf{fv\text{-}size}}$ is not polynomially bounded.
- The number of $\Rightarrow_{nd}$-steps is not polynomially bounded.

We illustrate the last two reasons in Example 26.

Example 26. Consider a signature with sorts $\{\iota_0, \dots, \iota_n\}$ where each sort has a single constructor: $\mathsf{c}_0 : \iota_0$ and $\mathsf{c}_{i+1} : \iota_i \times \iota_i \to \iota_{i+1}$ for all $i < n$. Then every ι_i contains exactly one ground term t_i, which is a full binary tree of depth i, i.e., $|x : \iota_i| = |t_i| = 2^i - 1$ and hence $|pp|_{\mathsf{fv\text{-}size}} \geq 2^n - 1$ whenever pp contains a variable of sort ι_n.

Next consider a TRS with defined symbol $\mathsf{f} : \iota_n \times \iota_n \to \iota_0$ which only contains a single left-hand side $\mathsf{f}(x, x)$. Pattern completeness of this TRS is encoded as the pattern problem $pp = \{\{y \sim x, z \sim x\}\}$ where all three variables are of sort ι_n. Then every reduction to normal form will have the shape $pp \Rightarrow_{nd}^{\geq 2^n} \{\{t_n \sim x, t_n \sim x\}\} \Rightarrow_{nd}^{*} \top_{pp}$.

Example 26 shows that the algorithm is actually using exponential time and space. In order to solve this problem, we will illustrate in the upcoming section, how we can modify the inference rules further in order to arrive at an algorithm to decide pattern completeness for arbitrary inputs, which again is contained in the complexity class co-NP.

5. Pattern completeness – the general case in co-NP

The main problem of the high complexity of the current non-deterministic algorithm $\Rightarrow_{nd}$ is the (instantiate') rule in Definition 22. As it was shown in Example 19, in general one needs to perform some instantiation, but on the other hand, as Example 26 illustrates, a full instantiation to constructor ground form is too expensive.

In this section we will alter the (instantiate') rule in a way that it is applied in a more restrictive way, that ultimately leads to a non-deterministic polynomial time algorithm.

This new algorithm will be split into four phases, where we will generate intermediate pattern problems that have a special form.

- The first phase resolves the term structure of patterns. After this phase, all matching atoms are of form $t \sim x$ with $x \in \mathcal{X}$.
- The second phase resolves the term structure of t above, up to constructor terms of finite sorts. This will require a generalized version of the (inf-diff) rule.
- The third phase eliminates those constructors, so that all matching atoms are of form $y \sim x$, where y is a variable of finite sort. The restricted version of (instantiate') will be included in this phase.
- The forth phase eventually encodes completeness of the resulting problem into a validity statement of a formula, with respect to a logic where validity is known to be decidable in co-NP.

We will illustrate the phases on the following challenging example.

Example 27. We consider a TRS with the same sorts and constructors as in Example 26 and a defined symbol $\mathsf{f} : \iota_0 \times \cdots \times \iota_n \to \iota_0$. The TRS has left-hand sides $\mathsf{f}(x_0, \mathsf{c}_1(x_0, x_0), \_, \dots, \_), \dots, \mathsf{f}(\_, \dots, \_, x_{n-1}, \mathsf{c}_n(x_{n-1}, x_{n-1}))$, where each underscore represents a fresh variable, and the aim is to check whether the TRS is pattern complete, i.e., the initial pattern problem is:

$$\begin{aligned} pp_0 = \{&\{\mathsf{f}(y_0, \dots, y_n) \sim \mathsf{f}(x_0, \mathsf{c}_1(x_0, x_0), \_, \dots, \_)\},\\ &\{\mathsf{f}(y_0, \dots, y_n) \sim \mathsf{f}(\_, x_1, \mathsf{c}_2(x_1, x_1), \_, \dots, \_)\},\\ &\quad\vdots\\ &\{\mathsf{f}(y_0, \dots, y_n) \sim \mathsf{f}(\_, \dots, \_, x_{n-1}, \mathsf{c}_n(x_{n-1}, x_{n-1}))\}\}. \end{aligned}$$

5.1. Phase one: From pattern problems to simplified pattern problems

First of all, we will switch to simplified pattern problems. These simplified problems correspond to pattern problems where in all matching atoms $t \sim \ell$ the pattern ℓ is a variable. Since variables always match, we can basically ignore the concrete variable names, and just collect the matchees of each variable in a separate equivalence class.

For instance, the matching problem $\{t_1 \sim x, t_2 \sim y, t_3 \sim x, t_4 \sim x, t_5 \sim y\}$ can equivalently be represented as $\{\{t_1, t_3, t_4\}, \{t_2, t_5\}\}$ where all patterns have been dropped.

Definition 28 (Simplified Form and Simplified Pattern Problem)**.**
A matching problem mp is in *simplified form* iff $mp \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{X}$. A pattern problem is in simplified form iff all its matching problems are in simplified form.

A *simplified* matching problem is a finite set mp of equivalence classes, and each equivalence class e is a non-empty finite set of matchee terms of the same sort. A *simplified* pattern problem is a finite set of simplified matching problems. A simplified matching problem mp is *complete* w.r.t. a constructor ground substitution $\sigma : \mathcal{V} \to \mathcal{T}(C)$ if $t_i\sigma = t_j\sigma$ for all $e \in mp$ and all $t_i, t_j \in e$. A simplified pattern problem pp is complete if for each $\sigma : \mathcal{V} \to \mathcal{T}(C)$ there is some $mp \in pp$ such that mp is complete w.r.t. σ. The variables of a simplified pattern problem are defined as $\mathsf{Var}(pp) = \bigcup_{t \in e \in mp \in pp} \mathsf{Var}(t)$.

Definition 29 (Transformation to Simplified Pattern Problems)**.**
Given a matching problem mp in simplified form

$$\{t_{11} \sim x_1, \dots t_{1n_1} \sim x_1, \dots, t_{k1} \sim x_k, \dots t_{kn_k} \sim x_k\}$$

where x_1, ..., x_k are distinct variables, we define its corresponding simplified matching problem as

$$\{\{t_{11}, \dots, t_{1n_1}\}, \dots \{t_{k1}, \dots, t_{kn_k}\}\}.$$

The simplified pattern problem corresponding to a pattern problem in simplified form is obtained by applying the above transformation on each matching problem.

It is easy to see that the switch to simplified pattern problems does not alter pattern completeness.

Lemma 30. *If pp is in simplified form, then completeness of pp is equivalent to completeness of its corresponding simplified pattern problem.*

Note that we already have a polynomial time algorithm to start from arbitrary pattern problems and arrive at simplified pattern problems.

Lemma 31 (Phase One: From Pattern Problems to Simplified Pattern Problems)**.** *Consider $\Rightarrow_{nd}$ for the linear case, i.e., Definition 6.*

- *If $pp \Rightarrow^{!}_{nd} pp'$, then $pp' = \top_{pp}$ or pp' is in simplified form.*
- *Pattern problem pp is complete iff the simplified pattern problem of pp' is complete for every pp' with $pp \Rightarrow^{!}_{nd} pp'$.*

Example 32. In this phase, pp_0 of Example 27 is transformed as follows:

$$\begin{aligned}
pp_0 \Rightarrow^{*}_{nd}\ & \{\{y_0 \sim x_0, y_1 \sim \mathsf{c}_1(x_0,x_0)\},\\
& \{y_1 \sim x_1, y_2 \sim \mathsf{c}_2(x_1,x_1)\},\\
& \{y_2 \sim x_2, y_3 \sim \mathsf{c}_3(x_2,x_2)\}, \dots\}\\
\Rightarrow_{nd}\ & \{\{y_0 \sim x_0, \mathsf{c}_1(z_1,z_2) \sim \mathsf{c}_1(x_0,x_0)\},\\
& \{\mathsf{c}_1(z_1,z_2) \sim x_1, y_2 \sim \mathsf{c}_2(x_1,x_1)\},\\
& \{y_2 \sim x_2, y_3 \sim \mathsf{c}_3(x_2,x_2)\}, \dots\}\\
\Rightarrow_{nd}\ & \{\{y_0 \sim x_0, z_1 \sim x_0, z_2 \sim x_0\},\\
& \{\mathsf{c}_1(z_1,z_2) \sim x_1, y_2 \sim \mathsf{c}_2(x_1,x_1)\},\\
& \{y_2 \sim x_2, y_3 \sim \mathsf{c}_3(x_2,x_2)\}, \dots\}\\
\Rightarrow_{nd}\ & \{\{y_0 \sim x_0, z_1 \sim x_0, z_2 \sim x_0\},\\
& \{\mathsf{c}_1(z_1,z_2) \sim x_1, \mathsf{c}_2(z_3,z_4) \sim \mathsf{c}_2(x_1,x_1)\},\\
& \{\mathsf{c}_2(z_3,z_4) \sim x_2, y_3 \sim \mathsf{c}_3(x_2,x_2)\}, \dots\}\\
\Rightarrow_{nd}\ & \{\{y_0 \sim x_0, z_1 \sim x_0, z_2 \sim x_0\},\\
& \{\mathsf{c}_1(z_1,z_2) \sim x_1, z_3 \sim x_1, z_4 \sim x_1\},\\
& \{\mathsf{c}_2(z_3,z_4) \sim x_2, y_3 \sim \mathsf{c}_3(x_2,x_2)\}, \dots\}\\
\Rightarrow^{*}_{nd}\ & \{\{y_0 \sim x_0, z_1 \sim x_0, z_2 \sim x_0\},\\
& \{\mathsf{c}_1(z_1,z_2) \sim x_1, z_3 \sim x_1, z_4 \sim x_1\},\\
& \{\mathsf{c}_2(z_3,z_4) \sim x_2, z_5 \sim x_2, z_6 \sim x_2\}, \dots\}
\end{aligned}$$

and this final pattern problem in simplified form is compressed into the simplified pattern problem

$$pp_1 := \{\{\{y_0, z_1, z_2\}\}, \{\{\mathsf{c}_1(z_1,z_2), z_3, z_4\}\}, \{\{\mathsf{c}_2(z_3,z_4), z_5, z_6\}\}, \dots\}$$

From now on we just process simplified pattern problems, and therefore often drop the explicit mentioning of the attribute "simplified".

5.2. Phase two: From simplified pattern problems to finite constructor form

The second phase aims at eliminating all occurrences of infinite sorts and also all occurrences of terms that are not constructor terms. This is made formally in the upcoming definition.

Definition 33 (Finite Constructor Form)**.** A matching problem mp is in *finite constructor form* if for all $t \in e \in mp$, $t \in \mathcal{T}(C,\mathcal{V})$, and all variables of t are of a finite sort. A pattern problem pp is in finite constructor form if all matching problems $mp \in pp$ are.

We use the following transformation rules to convert arbitrary pattern problems into finite constructor form. To this end, we first extend the notion of infinite variable difference to simplified matching problems: mp has such a difference iff there is some $e \in mp$ and there are t and t' where $\{t,t'\} \subseteq e$ and t and t' have an infinite variable difference.

Definition 34 (Transformation to Finite Constructor Form)**.** We define $\overset{s}{\to}$ as relation from simplified matching problems to extended simplified matching problems, i.e., where in the latter problems $\bot_{mp}$ is added. It consists of the following rules.

$$\{\{f(t_{11},\dots,t_{1n}),\dots,f(t_{k1},\dots,t_{kn})\},e_1,\dots,e_m\} \overset{s}{\to} \{\{t_{11},\dots,t_{k1}\},\dots,\{t_{1n},\dots,t_{kn}\},e_1,\dots,e_m\} \quad \text{if } k>1 \quad \text{(decompose')}$$

$$\{\{t,t',\dots\},\dots\} \overset{s}{\to} \bot_{mp} \quad \text{if } t \text{ and } t' \text{ clash} \quad \text{(clash")}$$

$$\{\{t\},e_1,\dots,e_m\} \overset{s}{\to} \{e_1,\dots,e_m\} \quad \text{(unique)}$$

$$\{\{x,t,\dots\},\dots\} \overset{s}{\to} \bot_{mp} \quad \text{if } x \in \mathcal{V} \text{ and } t \notin \mathcal{T}(C,\mathcal{V}) \quad \text{(no-constructor)}$$

We further define $\overset{s}{\Rightarrow}$ to work on simplified pattern problems where—similarly to $\Rightarrow$—a set of pattern problems is returned.

$$\{mp\} \uplus pp \overset{s}{\Rightarrow} \{\{mp'\} \cup pp\} \quad \text{if } mp \overset{s}{\to} mp' \neq \bot_{mp} \qquad \text{(simp-mp')}$$

$$\{mp\} \uplus pp \overset{s}{\Rightarrow} \{pp\} \quad \text{if } mp \overset{s}{\to} \bot_{mp} \qquad \text{(remove-mp')}$$

$$\{\top_{mp}\} \uplus pp \overset{s}{\Rightarrow} \top_P \qquad \text{(success')}$$

$$pp \uplus pp' \overset{s}{\Rightarrow} \{pp'\} \quad \text{if } pp \neq \emptyset, \text{each } mp \in pp \text{ has an infinite variable difference and all variables in } pp' \text{ are of finite sort} \qquad \text{(inf-diff')}$$

Finally, we define $\overset{s}{\Rightarrow}_{nd}$ which makes non-deterministic choices: $pp \overset{s}{\Rightarrow}_{nd} pp'$ iff $pp \overset{s}{\Rightarrow} P$ and $pp' \in P$, and $pp \overset{s}{\Rightarrow}_{nd} \top_{pp}$ iff $pp \overset{s}{\Rightarrow} \top_P$.

Regarding the new rule (inf-diff'), we see that this is mainly a generalization of the (inf-diff) rule: (inf-diff) is obtained from (inf-diff') if once chooses $pp' = \emptyset$.

The polynomial complexity of the algorithm is easy to see. Since each $\overset{s}{\Rightarrow}_{nd}$-step removes at least one symbol, termination and a polynomial upper bound on the number of possible $\overset{s}{\Rightarrow}_{nd}$-steps are obvious.

Therefore, we here emphasize the other two key properties of $\overset{s}{\Rightarrow}_{nd}$: the soundness and the shape of normal forms.

Lemma 35 (Phase Two: Switching to Finite Constructor Form)**.**

- *If $pp \overset{s\,!}{\Rightarrow}_{nd} pp'$, then $pp' = \top_{pp}$ or pp' is in finite constructor form.*
- *If $pp \overset{s}{\Rightarrow} P$, then pp is complete iff every $pp' \in P$ is complete.*
- *pp is complete iff every pp' with $pp \overset{s\,!}{\Rightarrow}_{nd} pp'$ is complete.*

Proof. We first show that any normal form pp of $\overset{s}{\Rightarrow}_{nd}$ must be in finite constructor form, so assume that pp is in normal form.

First consider some arbitrary $mp \in pp$. Since (unique) is not applicable, each $e \in mp$ must have at least two elements. Further, since (clash") and (decompose') are not applicable, there must be at least one variable in e. Consequently, all terms in e must be constructor terms, since otherwise (no-constructor) would be applicable. Hence, all equivalence classes in pp just contain constructor terms and at least one variable.

In order to show that all variables in pp are of finite sort, assume to the contrary that this would not be the case. Then we can split pp into $pp_f \uplus pp_{\neg f}$ where pp_f are those matching problems of pp that only contain variables of finite sort. We now show that $pp = pp_{\neg f} \cup pp_f \overset{s}{\Rightarrow} \{pp_f\}$ via (inf-diff') which is a contradiction to pp being in normal form. In order to validate this step, we focus on proving that each $mp \in pp_{\neg f}$ has an infinite variable difference. This can be seen as follows. Since $mp \in pp_{\neg f}$ there must be some $t \in e \in mp$ and $x \in \mathsf{Var}(t)$ such that x has an infinite sort. From the considerations above, we further know that t is a constructor term, and hence also t itself has an infinite sort, as it has x as subterm. Hence, all terms in e are of infinite sort. As argued above, each e consists of at least two terms and contains at least one variable, so we know $\{y, t'\} \subseteq e$ for some variable y and term t such that $y \neq t'$ and both y and t' are of infinite sort. Hence, y and t' differ in variable y and we have shown the existence of an infinite variable difference.

Next we consider the statement that $\overset{s}{\Rightarrow}$ preserves pattern completeness. This is obvious for several of the rules, and we will just consider two rules in detail.

For the (no-constructor) rule we want to mention that it is crucial to have a variable contained in the equivalence class. For instance, $\{\{f(x), f(y)\}\}$ might be complete for some $\sigma : \mathcal{V} \to \mathcal{T}(C)$, even if $f \notin C$. However, if x is a variable and $t \notin \mathcal{T}(C, \mathcal{V})$, then $x\sigma \neq t\sigma$ since $x\sigma \in \mathcal{T}(C)$ whereas $t\sigma \notin \mathcal{T}(C)$, so any matching problem containing $\{x, t, \dots\}$ as equivalence class cannot be complete for any $\sigma : \mathcal{V} \to \mathcal{T}(C)$.

Although the (inf-diff') rule is a generalization of the (inf-diff) rule, the proof of (inf-diff) reveals, that exactly the same construction also works out to prove soundness of (inf-diff'). The reason is that the major reasoning step in (inf-diff) is done by showing that σ can always be adjusted on the variables of infinite sort so that all infinite variable differences can be converted into clashes within pp. And since this adjustment has no effect on pp' if all variables in pp' are of finite sort, one can just remove pp. □

The following example shows that the generalization of (inf-diff) to (inf-diff') is necessary to obtain the desired shape of normal forms.

Example 36. Consider x_1, x_2 having finite sort, y_1, y_2 having infinite sort, and $\mathsf{f} \in C$. The following pattern problem would be a normal form if one would require $pp' = \emptyset$ in (inf-diff'), but it is not in finite constructor form.

$$\{\{\{x_1, x_2\}\}, \{\{\mathsf{f}(y_1), y_2\}\}\}$$

However, the problem is simplified to $\{\{\{x_1, x_2\}\}\}$, a problem that is in finite constructor form, and the simplification is performed via (inf-diff') in its general form.

Concerning the running example, we see that pp_1 in Example 32 is already in finite constructor form, so no steps are performed here.

5.3. Phase three: From finite constructor form to finite variable form

The third phase will get rid of all the constructors in a matching problem, so that afterwards just variables remain. For instance, as input one might have pattern problems that include equivalence classes of the form $\{c(x,y), z, c(b,x)\}$ for constructors b, c and variables x, y, z of finite sorts, and the aim is to obtain pattern problems where all terms in an equivalence class are variables. The latter kind of problems are named finite variable forms.

Definition 37 (Finite Variable Form)**.** A matching problem mp is in *finite variable form* if each $e \in mp$ contains only variables of the same finite sort. A pattern problem pp is in finite variable form if all matching problems $mp \in pp$ are.

In this phase we will add again the equivalent of the (instantiate') rule for simplified pattern problems. However, we will strongly limit its applicability, so that we can control the growth in the number of function symbols. To be able to perform this limitations we add a new non-deterministic (split) rule, that will guess where the incompleteness of some matching problem will materialize.

Definition 38 (Transformation to Finite Variable Form)**.** We extend the inference rules of Definition 34 by the following ones.

$$\{\{\{t,t'\} \uplus e\} \uplus mp\} \uplus pp \overset{s}{\Rightarrow} \{\{\{\{t,t'\}\}\} \cup pp, \{\{\{t'\} \cup e\} \cup mp\} \cup pp\} \quad \text{if exactly one of } t \text{ and } t' \text{ is a variable, and } e \neq \emptyset \text{ or } mp \neq \emptyset \quad \text{(split)}$$

$$pp \overset{s}{\Rightarrow} \mathsf{Inst}(pp, x) \quad \text{if } \{\{x,t\}\} \in pp, x \text{ has a finite sort, and } t \text{ is not a variable} \quad \text{(instantiate")}$$

Here, for a simplified 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 $pp\sigma_{x,c}$ for each $c : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in C$, where $pp\sigma_{x,c}$ is obtained from pp by applying $\sigma_{x,c} = [x \mapsto c(x_1, \ldots, x_n)]$ on each term within pp, for distinct fresh variables $x_1 : \iota_1, \ldots, x_n : \iota_n \in \mathcal{V}$.

Let us briefly explain both new rules. Rule (instantiate") is indeed a restricted version of (instantiate') in the way that instantiations are only allowed, if the conflicting variable occurs at the root, the conflicting term is not a variable itself, and moreover, there are no other terms in the matching problem that get instantiated. The latter two facts will be crucial to obtain non-deterministic polynomial complexity.

Since the restricted (instantiate") rule on its own is often not applicable, one has to use the (split) rule in a preprocessing step: whenever one has some matching problem $\{\{x,t\} \cup e\} \cup mp$ that would permit an application of (instantiate'), then one has to non-deterministically choose via (split) whether one wants to try to show that the equivalence of x and t can be violated by separating the matching problem $\{\{x,t\}\}$ that can then be instantiated by (instantiate"), or one can forget about the equivalence of x and t and search for incompleteness in the remaining problem, where either x or t is removed from the equivalence class.

Theorem 39 (Phase Three: Switching to Finite Variable Form)**.**

- *If* $pp \overset{s}{\Rightarrow}{}^{!}_{nd} pp'$*, then* $pp' = \top_{pp}$ *or* pp' *is in finite variable form.*
- *If* $pp \overset{s}{\Rightarrow} P$*, then* pp *is complete iff every* $pp' \in P$ *is complete.*
- pp *is complete iff every* pp' *with* $pp \overset{s}{\Rightarrow}{}^{!}_{nd} pp'$ *is complete.*
- *The number of* $\overset{s}{\Rightarrow}_{nd}$*-steps is polynomially bounded, and hence* $\overset{s}{\Rightarrow}_{nd}$ *is terminating.*

Proof. Let us first consider the possible normal forms, so assume that pp is in normal form, but not in finite variable form, and we will derive a contradiction. By Lemma 35 we know that pp is in finite constructor form and pp must contain at least one term t that is not a variable. So, there is a chain $t \in e \in mp \in pp$. From the proof of Lemma 35 we further know that e contains at least one variable x. Hence $\{x,t\} \subseteq e$. If $mp = \{\{x,t\}\}$ then (instantiate") is applicable. Otherwise, (split) is applicable. In both cases, we get a contradiction to pp being in normal form.

That the extended version of $\overset{s}{\Rightarrow}$ still preserves completeness is straightforward: (instantiate") is an instance of (instantiate'); and for (split) one can argue as follows for any $\sigma : \mathcal{V} \to \mathcal{T}(C)$ and $e = \{t_1, \ldots, t_n\}$.

$\{\{t,t'\} \uplus e\} \uplus mp$ is complete for σ
iff $t\sigma = t'\sigma = t_1\sigma = \cdots = t_n\sigma$ and mp is complete forσ
iff $t\sigma = t'\sigma$ and ($t'\sigma = t_1\sigma = \cdots = t_n\sigma$ and mp is complete for σ)
iff $\{\{t,t'\}\}$ is complete for σ and $\{\{t'\} \cup e\} \cup mp$ is complete for σ

The real challenge is proving the polynomial complexity bound. To this end, we use a similar measure as in Definitions 10 and 12. To be more precise, we adjust $\mathsf{tsyms}(pp)$ for simplified pattern problems in the obvious way, i.e., $\mathsf{tsyms}(pp) = \bigcup_{t \in e \in mp \in pp} \mathsf{syms}(t)$, and in this way also obtain $|pp|_{\mathsf{tsize}}$ and $|pp|_{\mathsf{dup}}$ for simplified pattern problems. The key point is now to switch from $|\cdot|_{\mathsf{fv\text{-}size}}$ to $|\cdot|_{\mathsf{fv\text{-}depth}}$ where the latter measure is based on the maximum term *depth* instead of the maximum term *size* for terms of finite sort: the term depth is bounded by $|C|$, whereas the term size can be exponential. More formally: For a finite sort ι, we define $|\iota|_{\mathsf{max\text{-}depth}}$ as $\max\{\mathsf{depth}(t) \mid t : \iota \in \mathcal{T}(C)\}$ and we define:

$$|pp|_{\mathsf{fv\text{-}depth}} = \sum_{mp \in pp} \max\{|\iota|_{\mathsf{max\text{-}depth}} \mid x \in \mathsf{Var}(t) \text{ of a finite sort } \iota, t \in e \in mp\}$$

The overall measure for simplified pattern problems is defined as

$$|pp|_{\mathsf{spat}} := |pp|_{\mathsf{fv\text{-}depth}} \cdot (|pp|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp|_{\mathsf{tsize}}$$

where again, m is the maximum arity of the symbols in $\mathcal{C}$. One can show that this polynomially bounded measure is strictly decreasing with each $\overset{s}{\Rightarrow}_{nd}$-step, in the same way as we proved such a statement for $\Rightarrow_{nd}$, provided that $|pp|_{\mathsf{fv\text{-}depth}} > |pp'|_{\mathsf{fv\text{-}depth}}$ holds for $pp \overset{s}{\Rightarrow}_{nd} pp'$ by (instantiate").

Hence, it only remains to show that an application of the (instantiate")-rule indeed strictly decreases $|\cdot|_{\mathsf{fv\text{-}depth}}$. To this end, it is easy to see that an instantiation of a finite variable x will never increase the maximum term depth of the sorts of the variables, so a weak decrease is easy to obtain when switching from mp to $mp\sigma_{x,c}$. Moreover, in the matching problem $\{\{x,t\}\}$ that triggered the (instantiate")-rule, we see that the sort of each variable in $x\sigma_{x,c}$ and $t\sigma_{x,c}$ has a strictly smaller depth measure than the sort of x and t itself. In total, this leads to a strict decrease: $|pp|_{\mathsf{fv\text{-}depth}} > |pp\sigma_{x,c}|_{\mathsf{fv\text{-}depth}}$. □

The next theorem provides a combined complexity result on the first three phases. It shows an upper bound on the number of steps, including a size bound on intermediate problems.

Theorem 40 (Complexity to Reach Finite Variable Form). *Let pp and pp_1 be pattern problems, and pp'_1 and pp_2 be simplified pattern problems. We further assume*

- $pp \Rightarrow_{nd}^{n_1} pp_1$ *and* pp_1 *is in normal form w.r.t.* $\Rightarrow_{nd}$,
- pp'_1 *is the translation of* pp_1 *into simplified form, and*
- $pp'_1 \overset{s}{\Rightarrow}_{nd}^{n_2} pp_2$.

Then

$$n_1 + n_2 + |pp_2|_{\mathsf{tsize}} \leq (|\mathcal{C}| \cdot |pp| + |pp|_{\mathsf{size}}) \cdot (|pp|_{\mathsf{size}} \cdot \mathsf{m} + 2)$$

and

$$|pp_2| \leq |pp|$$

where as before, m *is the maximum arity of constructors in* $\mathcal{C}$.

Proof. The second inequality is easy to obtain, since whenever $pp_a \Rightarrow_{nd} pp_b$ or $pp_a \overset{s}{\Rightarrow}_{nd} pp_b$ then $|pp_b| \leq |pp_a|$.

For proving the first inequality we choose $\alpha = |\mathcal{C}| \cdot |pp|$ and derive

$$\begin{aligned}
& n_1 + n_2 + |pp_2|_{\mathsf{tsize}} \\
\leq\ & n_1 + n_2 + |pp_2|_{\mathsf{spat}} \\
\leq\ & n_1 + \left|pp'_1\right|_{\mathsf{spat}} \\
=\ & n_1 + \left|pp'_1\right|_{\mathsf{fv\text{-}depth}} \cdot (\left|pp'_1\right|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + \left|pp'_1\right|_{\mathsf{tsize}} \\
\leq\ & n_1 + (|\mathcal{C}| \cdot |pp'_1|) \cdot (\left|pp'_1\right|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + \left|pp'_1\right|_{\mathsf{tsize}} \\
=\ & n_1 + (|\mathcal{C}| \cdot |pp_1|) \cdot (|pp_1|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp_1|_{\mathsf{tsize}} \\
\leq\ & n_1 + \alpha \cdot (|pp_1|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp_1|_{\mathsf{tsize}} \\
\leq\ & n_1 + (\alpha + |pp_1|_{\mathsf{diff}}) \cdot (|pp_1|_{\mathsf{dup}} \cdot \mathsf{m} + 1) + |pp_1|_{\mathsf{tsize}} \\
=\ & n_1 + |pp_1|_{\alpha,\mathsf{pat}} \\
\leq\ & |pp|_{\alpha,\mathsf{pat}} \\
\leq\ & (\alpha + |pp|_{\mathsf{size}}) \cdot (|pp|_{\mathsf{size}} \cdot \mathsf{m} + 2)
\end{aligned}$$

by using Lemma 13, Theorem 15, the approximation $\left|pp'_1\right|_{\mathsf{fv\text{-}depth}} \leq |\mathcal{C}| \cdot |pp'_1|$, and the fact that every $\overset{s}{\Rightarrow}_{nd}$ step results in a decrease w.r.t. $|\cdot|_{\mathsf{spat}}$. □

Note that without the splitting and the restriction on the (instantiate") rule we would not obtain a polynomial time algorithm: For a relaxed version where instantiation is allowed as soon as some equivalence class contains two terms x and t where t is not a variable, one obtains an exponential time and exponential space algorithm, as illustrated by Example 41.

Example 41. If we continue to process Example 32 with the relaxed rule to obtain a finite variable form, we see an exponential run with exponentially large terms. We underline those parts where the changes take place.

$$\begin{aligned}
pp_1 = \{&\{\{y_0, z_1, z_2\}\},\\
&\{\{\mathsf{c}_1(z_1, z_2), \underline{z_3}, z_4\}\},
\end{aligned}$$

$$
\begin{aligned}
&\{\{\mathsf{c}_2(z_3, z_4), z_5, z_6\}\},\\
&\{\{\mathsf{c}_3(z_5, z_6), z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\},\\
&\{\{\mathsf{c}_1(z_1, z_2), \mathsf{c}_1(u_1, u_2), \underline{z_4}\}\},\\
&\{\{\mathsf{c}_2(\mathsf{c}_1(u_1, u_2), z_4), z_5, z_6\}\},\\
&\{\{\mathsf{c}_3(z_5, z_6), z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\},\\
&\{\{\underline{\mathsf{c}_1}(z_1, z_2), \underline{\mathsf{c}_1}(u_1, u_2), \underline{\mathsf{c}_1}(u_3, u_4)\}\},\\
&\{\{\mathsf{c}_2(\mathsf{c}_1(u_1, u_2), \mathsf{c}_1(u_3, u_4)), z_5, z_6\}\},\\
&\{\{\mathsf{c}_3(z_5, z_6), z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\},\\
&\{\{z_1, u_1, u_3\}, \{z_2, u_2, u_4\}\},\\
&\{\{\underline{\mathsf{c}_2(\mathsf{c}_1(u_1, u_2), \mathsf{c}_1(u_3, u_4)), z_5, z_6}\}\},\\
&\{\{\mathsf{c}_3(z_5, z_6), z_7, z_8\}\}, \ldots\}\\
\overset{s\ *}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\},\\
&\{\{z_1, u_1, u_3\}, \{z_2, u_2, u_4\}\},\\
&\{\{u_1, u_5, u_9\}, \{u_2, u_6, u_{10}\}, \{u_3, u_7, u_{11}\}, \{u_4, u_8, u_{12}\}\},\\
&\{\{\mathsf{c}_3(\mathsf{c}_2(\mathsf{c}_1(u_5, u_6), \mathsf{c}_1(u_7, u_8)), \mathsf{c}_2(\mathsf{c}_1(u_9, u_{10}), \mathsf{c}_1(u_{11}, u_{12}))), z_7, z_8\}\}, \ldots\}
\end{aligned}
$$

In contrast, there is no such exponential behavior with the strong restrictions on (instantiate") and (split).

Example 42. We now continue to process Example 32 with the correct rules of the third phase to obtain pp_3, a problem in finite variable form. Note that we only explore one possible way in the non-deterministic algorithm, and do not provide all alternatives. In the example one can see, that where both arguments of some c_i have been expanded in Example 41, here only one of the arguments of each c_i can be expanded. So, the algorithm basically expands single paths in terms, not full terms.

$$
\begin{aligned}
pp_1 = \ &\{\{\{y_0, z_1, z_2\}\}, \{\{\mathsf{c}_1(z_1, z_2), \underline{z_3}, z_4\}\},\\
&\{\{\mathsf{c}_2(z_3, z_4), z_5, z_6\}\}, \{\{\mathsf{c}_3(z_5, z_6), z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{\mathsf{c}_1(z_1, z_2), \underline{z_4}\}\},\\
&\{\{\mathsf{c}_2(z_3, z_4), z_5, z_6\}\}, \{\{\mathsf{c}_3(z_5, z_6), z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{\underline{\mathsf{c}_1}(z_1, z_2), \underline{\mathsf{c}_1}(u_1, u_2)\}\},\\
&\{\{\mathsf{c}_2(z_3, \mathsf{c}_1(u_1, u_2)), z_5, z_6\}\}, \{\{\mathsf{c}_3(z_5, z_6), z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{\{z_1, u_1\}, \{z_2, u_2\}\},\\
&\{\{\mathsf{c}_2(z_3, \mathsf{c}_1(u_1, u_2)), z_5, \underline{z_6}\}\}, \{\{\mathsf{c}_3(z_5, z_6), z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{z_1, u_1\}, \{z_2, u_2\}\},\\
&\{\{\mathsf{c}_2(z_3, \mathsf{c}_1(u_1, u_2)), z_5\}\}, \{\{\underline{\mathsf{c}_3(z_5, z_6)}, z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{z_1, u_1\}, \{z_2, u_2\}\},\\
&\{\{\mathsf{c}_2(z_3, \mathsf{c}_1(u_1, u_2)), \underline{z_5}\}\}, \{\{z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{z_1, u_1\}, \{z_2, u_2\}\},\\
&\{\{\underline{\mathsf{c}_2}(z_3, \mathsf{c}_1(u_1, u_2)), \underline{\mathsf{c}_2}(u_3, u_4)\}\}, \{\{z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{z_1, u_1\}, \{z_2, u_2\}\},\\
&\{\{z_3, u_3\}, \{\underline{\mathsf{c}_1(u_1, u_2)}, u_4\}\}, \{\{z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{z_1, u_1\}, \{z_2, u_2\}\},\\
&\{\{z_3, u_3\}, \{\underline{u_4}\}\}, \{\{z_7, z_8\}\}, \ldots\}\\
\overset{s}{\Rightarrow}_{nd}\ &\{\{\{y_0, z_1, z_2\}\}, \{\{z_1, u_1\}, \{z_2, u_2\}\},
\end{aligned}
$$

$$\{\{z_3, u_3\}\}, \{\{z_7, z_8\}\}, \ldots\} =: pp_3$$

5.4. Phase four: Solving problems in finite variable form

It remains to deal with pattern problems pp in finite variable form, i.e., every $e \in mp \in pp$ consists of variables of the same finite sort. According to Definition 28 such a problem is pattern complete iff for every substitution $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C})$ there exists $mp \in pp$ such that $\{x_1, \ldots, x_n\} \in mp$ implies $x_1\sigma = \cdots = x_n\sigma$. At this point it is easy to see that the term structure is of no relevance any further, i.e., we can switch from term substitution $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C})$ to integer assignment $\alpha : \mathcal{V} \to \mathbb{Z}$. The only gap here is that the number of different terms of sort ι is $|\iota| = |\{t : \iota \in \mathcal{T}(\mathcal{C})\}|$, so one must constrain the integer assignments to take this bound into account.

Definition 43 (Translation to SMT Formula)**.** Let pp be some pattern problem in finite variable form such that $\mathsf{Var}(pp) = \{y_1 : \iota_1, \ldots, y_n : \iota_n\}$. We define the encoding $\psi(pp)$ of pp as

$$\psi(pp) := \left(\bigwedge_{k=1}^{n} 1 \le y_k \le |\iota_k|\right) \longrightarrow \left(\bigvee_{mp \in pp} \bigwedge_{\{x_1,\ldots,x_k\} \in mp} x_1 = \cdots = x_k\right).$$

To finalize the decision procedure, we just need some algorithm to decide the validity of formula $\psi(pp)$. Note that $\psi(pp)$ can be expressed in well known logics such as integer difference logic, linear integer arithmetic, or bit-vector arithmetic. It is well known that deciding validity in each of these three logics is co-NP complete.

Lemma 44 (Phase Four: Deciding Completeness via SMT Solving)**.** *Let pp be a pattern problem in finite variable form.*

- *Pattern problem pp is pattern complete iff $\psi(pp)$ is valid in $\mathbb{Z}$.*
- *Validity of $\psi(pp)$ is in co-NP.*

5.5. Optimizations

At this point we can combine all phases in order to arrive at a decision procedure for pattern completeness. Our implementation does exactly this, but it makes use of two further inference rules that have not yet been mentioned.

One of these inference rules handles *trivial* sorts (like *unit*), which are inhabited by exactly one ground constructor term. All equivalence classes of trivial sorts in each matching problem can immediately be deleted as all terms after applying some constructor ground substitution will always be the unique constructor ground term of that sort. The other trivial case is when the cardinality of ι is large, since then one can always find constructor ground instances which behave like injective functions on the variables of sort ι, and hence matching problems containing such sorts can be deleted.

Definition 45 (Optimization Rules Based on Cardinalities)**.** We add the following two rules for simplifying problems in finite constructor form.

$$\{e\} \uplus mp \overset{s}{\to} mp \qquad \text{if the terms in } e \text{ have sort } \iota \text{ and } |\iota| = 1 \qquad \text{(card-1)}$$

$$pp \uplus \{\{\{x_1, t_1\} \uplus e_1\} \uplus mp_1, \ldots, \{\{x_n, t_n\} \uplus e_n\} \uplus mp_n\} \overset{s}{\Rightarrow} \{pp\} \qquad \text{if } \{x_1, \ldots, x_n\} \text{ are variables of sort } \iota \text{ that are disjoint from } \mathsf{Var}(pp), |\{t_1, \ldots, t_n\}| < |\iota|, n > 0, \text{and } x_i \neq t_i \text{ for each } i \qquad \text{(card-large)}$$

Example 46. Consider again pp_1, the pattern problem that we obtained in Example 32 after running the first phase. If ι_0 has cardinality one, then so do all ι_i and (card-1) can trivially delete all equivalence problems and pattern completeness is concluded. On the other hand, if for instance ι_0 has two constructors, then the cardinalities grow exponentially, and one can apply (card-large) to delete most matching problems without the necessity to perform an expensive non-deterministic search in the third phase.

As explained above, the soundness of (card-large) follows from the fact that one can modify each $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C})$ on $x_1, \ldots, x_n$ into a substitution $\sigma' : \mathcal{V} \to \mathcal{T}(\mathcal{C})$ in a way that (i), $x_i\sigma' \neq t_i\sigma'$ for each i, and (ii), the modification has no impact on pp, i.e., $y\sigma = y\sigma'$ for each variable y that occurs in pp.

5.6. Computing cardinalities

Interestingly, the algorithm that we designed so far does not have non-deterministic polynomial time or polynomial space complexity: both time and space consumption can grow exponentially, with respect to the number of sorts. In particular this happens for Examples 26 and 27 in contrast to the discussion in Example 46. There, it was argued that all these TRSs should be easily solvable by the (card-1) and (card-large) rules, depending on the cardinality of ι_0.

The cause of this exponential behavior is neither an exponential number of rule applications, nor an exponential growth of intermediate pattern problems. The problem is caused by something that we completely ignored up to now, namely the size of the

numbers that occur during the computation, i.e., the cardinalities of the finite sorts. These cardinalities can easily be computed in a bottom-up way—starting with small sorts—using the following equation:

$$|\iota| = \sum_{c:\iota_1\times\ldots\times\iota_n\to\iota\in C} \prod_{i=1}^{n} |\iota_i|.$$

Note that for the signature in Examples 26 and 27 we obtain $|\iota_{i+1}| = |\iota_i|^2$ for all $i < n$. If we define ι_0 in a way that $|\iota_0| = 2$ then this results in $|\iota_n| = 2^{2^n}$, and hence the number $|\iota_n|$ has a binary representation of exponential size. Therefore, already the computation of $|\iota_n|$ will require exponential space: Hence a naive implementation of phase four without the (card-large) optimization exhibits an exponential complexity.

Our work-around to this problem is as follows. We first fix a sufficiently large number k: given some initial pattern problem pp_{init}, the value of k just needs to satisfy $k > |pp_{init}|$ and $k > 1$, so we choose $k = \max(2, |pp_{init}| + 1)$.

Then instead of using $|\iota|$ we compute $\min(k, |\iota|)$, a task that can be done in polynomial time via the following equation:

$$\min(k, |\iota|) = \min\left(k, \sum_{c:\iota_1\times\ldots\times\iota_n\to\iota\in C} \prod_{i=1}^{n} \min(k, |\iota_i|)\right).$$

Also the pattern completeness algorithm can easily be adjusted to use these pruned cardinalities: the test $|\iota| = 1$ from the (card-1) rule is equivalent to $\min(k, |\iota|) = 1$, since $k > 1$. Similarly, the test $|\{t_1, \ldots, t_n\}| < |\iota|$ in the (card-large) rule is equivalently expressed as $|\{t_1, \ldots, t_n\}| < \min(k, |\iota|)$ whenever $k > n$. The latter condition is always satisfied since $n \le |pp_{init}| < k$. The reason for inequality $n \le |pp_{init}|$ is the second inequality of Theorem 40.

Finally, there is the step of encoding pattern problems pp in finite variable form into SMT formulas, where the exact cardinalities are required. At this point there cannot be any large sorts present anymore, since variables of all these sorts will have been eliminated by (card-large) before. To be more precise, for every sort ι that still is present in the final pattern problem, we obtain $|\iota| \le |pp| \le |pp_{init}| < k$ by using the second inequality of Theorem 40 again, and hence $|\iota| = \min(k, |\iota|)$. This equality shows that $|\iota|$ is a number that can be obtained from $\min(k, |\iota|)$, and it further proves that $|\iota|$ is polynomially bounded in the input size. Therefore the SMT formula in Definition 43 has polynomial size, and it can be computed in polynomial time, too.

Overall, with this final optimization, we arrive at concluding that general pattern completeness is co-NP.

Theorem 47. *The algorithm of Section 5 decides pattern incompleteness in non-deterministic polynomial time.*

Recall that the theorem is stated using the global assumptions that terms and pattern problems respect the sorts, and that all sorts are inhabited.

6. Formalization and implementation

In the formalization we first describe our development on sorted term rewriting (Section 6.1), which is essential to develop the formalization of the decision procedures of Sections 4 and 5 in Isabelle.

Afterwards we consider the formalization of the decision procedure in Section 4 in detail. Here, we utilize an architecture that uses three different layers of abstraction. We start with an abstract set based formalization (Section 6.2), then refine it to a version based on multisets (Section 6.3), and finally arrive at an executable version that uses lists to represent matching and pattern problems (Section 6.4).

The same architecture is also used for the decision procedure of Section 5, and therefore, we only briefly describe the formalization of this part (Section 6.5).

The overall formalization consists of 20,056 lines of Isabelle. It is available for Isabelle 2025-2 in two AFP entries, one on sorted terms [18], and one on pattern completeness [19].

6.1. 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 [20].

```
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 ⇀ 's*, which is a synonym of *'a → 's option*. We just introduce the notation "*a : ι in A*" to mean that a sorted set *A :: 'a ⇀ 's* assigns its element *a :: 'a* a sort *ι :: 's*.

```
definition ... where "a : ι in A ≡ A a = Some ι"
```

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

We formalize sorted signatures also as partial maps:

```
type_synonym ('f,'s) ssig = "'f × 's list ⇀ 's"
```

and introduce the following notation:

definition ... **where** *"f : ıs → ı₀ in F ≡ F (f,ıs) = Some ı₀"*

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

"Var v : ı in 𝒯 (F,V) ⟷ v : ı in V"
"Fun f ts : ı₀ in 𝒯 (F,V) ⟷ (∃ıs. f : ıs → ı₀ in F ∧ ts :ₗ ıs in 𝒯 (F,V))"

Here, *as :ₗ ıs in A* denotes that the lists *as* and *ıs* have the same length and the *i*th element of *as* has the *i*th sort of *ıs*. Given two signatures *C* and *D*, the sorted set of basic terms is formalized as follows:

"Fun f ts : ı₀ in 𝒯_B(C,D,V) ⟷ (∃ıs. f : ıs → ı₀ in D ∧ ts :ₗ ıs in 𝒯 (C,V)"

We also introduce the notation ∅ for *Map.empty*, the partial map such that *∅ a = None* for any *a*. Then *𝒯 (F,∅)* denotes the sorted set of ground terms, where the type of variables is polymorphic. Often we are not interested in the type of variables of ground terms, so we write *𝒯 (F)* or *𝒯_B(C,D)* to fix the type to *unit*.

A *sorted map* from a sorted set *A* to a sorted set *B*, written *f :ₛ A → B* in Isabelle, is a map *f* such that *a : ı in A ⟹ f a : ı in B*. In particular, sorted maps of form *σ :ₛ X → 𝒯 (F,V)* are the sorted substitutions. The application of a substitution *σ* on a term *t* is already defined as *t · σ* in the library in the unsorted setting. We additionally provide facts such as

lemma *subst_hastype: "σ :ₛ X → 𝒯 (F,V) ⟹ t : ı in 𝒯 (F,X) ⟹ t · σ : ı in 𝒯 (F,V)"*

The formalization of when a term matches another is straightforward:

definition ... **where** *"l matches t = (∃ σ. t = l · σ)"*

and now we are ready to define the pattern completeness.

definition ... **where** *"pat_complete_lhss C D L = (∀t ∈ dom 𝒯_B(C,D). ∃l ∈ L. l matches t)"*

Here, *dom* denotes the *domain* of partial maps, i.e., the set of elements that are assigned sorts.

6.2. 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 ⇛.

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 → 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 ⊎-operators in the textual description into ∪-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 → and ⇛, illustrating the first two kinds of deviations. Here *insert a A* is Isabelle's notation for $\{a\} \cup A$.

inductive ... **where** *"mp →ₛ mp"*
| *"length ts = length ls ⟹ insert (Fun f ts, Fun f ls) mp →ₛ set (zip ts ls) ∪ mp"*
| *"x ∉ ⋃ (vars ‘ snd ‘ mp) ⟹ insert (t, Var x) mp →ₛ mp"*

inductive *mp_fail :: "('f,'v,'s)match_problem_set → bool"*
where *"(f,length ts) ≠ (g,length ls) ⟹ mp_fail (insert (Fun f ts, Fun g ls) mp)"*
| `(* further inference rule for clash' *)`

inductive ... **where** *"mp →ₛ mp' ⟹ insert mp pp ⇛ₛ {insert mp' pp}"*
| *"mp_fail mp ⟹ insert mp pp ⇛ₛ {pp}"*
| *"insert {} pp ⇛ₛ {}"*
| `(* further inference rules for inf-diff and instantiate *)`

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 C P* $\longleftrightarrow$ *pats_complete C P'*"

The most challenging rule was (inf-diff) 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.

6.3. 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 24 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

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

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

The reduction relation $\Rrightarrow$ on this layer is parameterized by a Boolean flag *improved* to choose the version presented in FSCD or the version introduced in Section 5. In either case, the partial correctness of $\Rrightarrow$ on this layer is obtained via that 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. Moreover, the multiset layer also contains a formal definition of $\Rightarrow_{nd}$ that deviates from the textual one since in the formal version the transition to $\top_{pp}$ is omitted.

We arrive at a formal version of Theorem 24 that looks quite similar to the textual one. Here, $\Rightarrow_{nd}$ in Isabelle refers to the multiset representation of $\Rightarrow_{nd}$, *SN* is strong normalization, i.e., termination, *pat_mset* converts from the multiset representation of pattern problems into the set representation, and *{#}* is the empty multiset.

theorem *SN_nd_pstep:* "*SN* $\Rightarrow_{nd}$"
theorem *nd_pstep:*
  assumes "¬ *improved*"
    and "*wf_pat (pat_mset p)*"
  shows "¬ *pat_complete (pat_mset p)* $\longleftrightarrow$ *p* $\Rightarrow_{nd}^{*}$ *{#}*"

We further obtain the results for relation $\Rrightarrow$, i.e., they look more like Theorem 17, though the linearity conditions are missing, since we are considering the algorithm in Section 4. Here, $\Rrightarrow$ in Isabelle refers to the multiset representation of $\Rrightarrow$, and *bottom_mset* is representing $\bot_P$ as the multiset $\{\emptyset\}$.

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

The partial correctness of the improved version is formalized as follows. Here, we do not directly decide pattern completeness but obtain pattern problems which are in finite constructor form. To be more precise, $\Rrightarrow$ with the activated *improved* flag implements phase one and two of Section 5. Below $\in$# is the multiset membership.

theorem *P_step_improved:*
  fixes *P :: "('f,'v,'s) pats_problem_mset"*
  assumes *"improved"*
    and *"infinite (UNIV :: 'v set)"*
    and *"wf_pats (pats_mset P)"*
    and *"(P,Q)* $\in \Rrightarrow^{!}$"
  shows *"pats_complete C (pats_mset P)* $\longleftrightarrow$ *pats_complete C (pats_mset Q)"*
    and *"p* $\in$# *Q* $\Longrightarrow$ *finite_constr_form_pat C (pat_mset p)"*

6.4. 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 atoms of form $x \sim f(\dots)$, i.e., those on which (instantiate) is applicable.
- We store another list of pairs $([t_1, \dots, t_n], x)$ such that the matching problem contains atoms $t_1 \sim x, \dots, t_n \sim x$, 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 (inf-diff). 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: $\{t \mid t : \iota \in \mathcal{T}(C)\} \neq \emptyset$. 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 $c : \iota_1 \times \cdots \times \iota_n \to \iota_0 \in C$ 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 also require a function that computes the finite sorts and their cardinalities. The algorithm maintains for each sort ι a mark signifying finiteness and $|\iota|$ when ι is marked. Initially all sorts are considered potentially infinite and cardinalities are unknown. If there is a potentially infinite sort ι such that all constructors $f : \iota_1 \times \cdots \times \iota_n \to \iota \in C$ of that sort have only finite input sorts $\iota_1, \dots, \iota_n$, then ι is marked as finite and the cardinality is updated to $\sum_{f : \iota_1 \times \cdots \times \iota_n \to \iota \in C} |\iota_1| \times \cdots \times |\iota_n|$. When there is no such sort anymore, exactly the finite sorts are marked and their cardinalities are computed. Interestingly, the dual approach (marking of infinite sorts whenever a recursive constructor is detected) is not so straightforward, 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_fscd*. The suffix FSCD is used, since this algorithm was already formalized in our FSCD paper [1].

theorem *decide_pat_complete_lhss_fscd:*
assumes *"decide_pat_complete_lhss_fscd 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 contains contradictory sort information. 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.

The formalization also contains *decide_pat_complete_linear_lhss*, a verified version of the algorithm for linear pattern problems. Its formalization is only present in the list layer, where the linearity preservation is performed during the refinement proof. The advantage of this algorithm over *decide_pat_complete_lhss_fscd* is that it is slightly faster, since (match) is applied without any runtime checks on the condition $\forall s \sim \ell \in mp.\ x \notin \mathsf{Var}(\ell)$.

6.5. Formalization of the co-NP algorithm

As already mentioned, by activating the parameter *improved*, we already have a formally verified implementation of the first two phases of the algorithm that was presented in Section 5.

The list-based implementation in the *improved* case is designed in a way that it takes a solver for problems in finite constructor form, and whenever it encounters such a problem, it just invokes the solver and then continues to process the remaining problems. Under the assumption that the solver is sound, one again gets a decision procedure with the same soundness guarantees as before.

The formalization also includes a verified solver for problems in finite constructor form, namely by formalizing the results of phase three, following the same structure as before: simplified matching and pattern problems and the corresponding algorithms are formalized for sets, multisets and lists. Moreover, also for phase three, the list-based implementation requires as parameter some solver, namely some SMT solver that is able to validity the formulas that are generated in phase four.[4]

The formalization further includes the main complexity results for the multiset layer, namely Theorem 40, including the result that the cardinalities of the sorts in the final pattern problems before the translation to SMT are bounded from above by the cardinality of the input pattern problem.

[4] In the sources, this solver is named "FIDL-solver": finite integer difference logic.

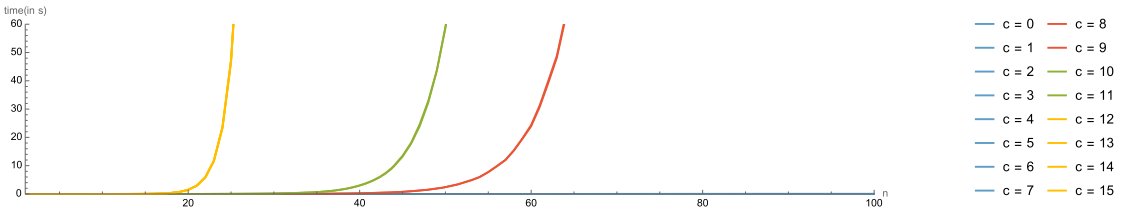


Fig. 1. Timing of our algorithm for each configuration $c \in \{0, \dots, 15\}$ on different sizes $n \in \{1, \dots, 100\}$ using a timeout of 60 s.

Finally, we provide a verified SMT solver for the fragment of formulas that have to be treated. Since this is a finite search problem, we implemented and verified a basic brute-force algorithm with some optimizations, and do not treat full integer difference logic where they may also be unbounded variables.

- We try to identify unit-clauses and then perform theory propagation. For example, if x was set to 5 and we have a unit clause $x \neq y$, then 5 is deleted from the set of possible values for y, and if there is only one value left, then also y is directly assigned, etc.
- We do a limited form of symmetry breaking: Initially, for each sort one can set one variable to 1 without loss of generality.

Plugging the various solvers together, we arrive at the function *decide_pat_complete_lhss*. It has the same soundness properties as *decide_pat_complete_lhss_fscd*, but implements the algorithm of Section 5.

7. Experiments

In this section we will investigate the efficiency of our decision procedure by experiments. We first restrict to linear pattern problems, and afterwards consider non-linear problems.

7.1. Linear pattern problems

In order to evaluate the efficiency of our decision procedure in Section 3 for linear pattern problems, we use the following set of left-linear TRSs. They are similar to test programs that are used to show exponential behavior of match compilers for functional programming languages [14, Example 6].

Example 48. We define TRS $\mathcal{R}_{c,n}$ for configurations $c \in \{0, \dots, 15\}$ and arbitrary 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.

$c = 0$	$c = 1$	$c = 2$	$c = 4$
g(F, _, F, _, F, _)	g(F, F, F, _, _, _)	g(F, _, F, _, F, _)	g(T, T, _, _, _, _)
g(T, T, _, _, _, _)	g(T, _, _, T, _, _)	g(T, T, _, _, _, _)	g(T, F, _, _, _, _)
g(T, F, _, _, _, _)	g(T, _, _, F, _, _)	g(_, _, T, T, _, _)	g(_, _, T, T, _, _)
g(_, _, T, T, _, _)	g(_, T, _, _, T, _)	g(_, _, _, _, T, T)	g(_, _, T, F, _, _)
g(_, _, T, F, _, _)	g(_, T, _, _, F, _)	g(T, F, _, _, _, _)	g(_, _, _, _, T, T)
g(_, _, _, _, T, T)	g(_, _, T, _, _, T)	g(_, _, T, F, _, _)	g(_, _, _, _, T, F)
g(_, _, _, _, T, F)	g(_, _, T, _, _, F)	g(_, _, _, _, T, F)	g(F, _, F, _, 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 $\mathcal{R}_{0,n}$:

g(F, _, F, _, … , F, _) or g(_, F, _, F, … , _, F) or g(F, F, … , F, _, _, … , _) or g(_, _, … , _, F, F, … , 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. Fig. 1.

Choosing $c \in \{0, \dots, 7\}$ results in a low execution time, where the corresponding blue line in Fig. 1 is not distinguishable from the x-axis: for instance, deciding pattern completeness of $\mathcal{R}_{c,100}$ is finished within 0.06 s. However, for $c \in \{8, \dots, 15\}$ 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.

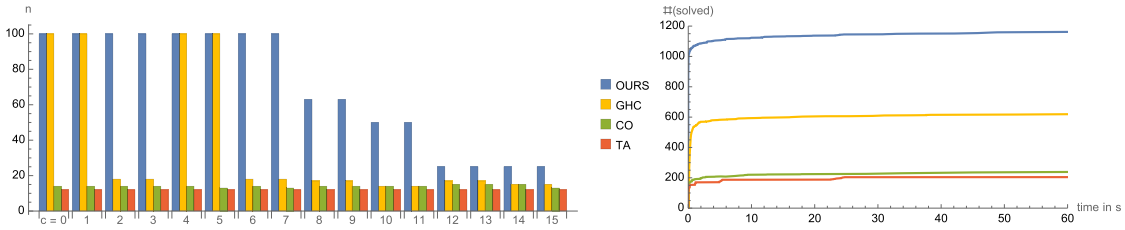


Fig. 2. Left: Increase n for each $c \in \{0, \ldots, 15\}$ until $n = 100$ or 60 s timeout is reached. Right: Number of solved $\mathcal{R}_{c,n}$ instances for $(c, n) \in \{0, \ldots, 15\} \times \{1, \ldots, 100\}$ within time limit.

- 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.[5]
- CO: we run the complement algorithm on the TRSs, taking the implementation that is available in the ground confluence prover AGCP [4]. To be more precise, we execute the function `Gcr.patCovered` from the AGCP source on the various inputs.
- 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 $\mathsf{g}(\mathsf{bool}, \ldots, \mathsf{bool}) \to \mathsf{accept}$) 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 transition for each rule, e.g., $\mathsf{g}(\mathsf{bool}, \mathsf{bool}, \mathsf{bool}, \mathsf{true}, \mathsf{false}) \to \mathsf{accept}$ 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 [21] to decide this inclusion property via its `TA.Subset.subset` algorithm.

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 s timeout when handling $\mathcal{R}_{c,n}$. In Fig. 2 we display the maximal values of n (left) and the cumulative solved instances plot in the style of SAT-competition [22] (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.

7.2. Non-linear pattern problems

In this section we compare the original algorithm of the FSCD paper from Section 4 (FSCD) with the new algorithm from Section 5 (NEW). In order to evaluate these algorithms we use three different kinds of examples, each in two variations. Each positive variant is pattern complete and each negative variant is not. All of the examples are parameterized by some number n, leading to TRSs $\mathcal{R}_n$.

The first two examples are the TRSs of Examples 26 and 27. Here, the positive variants are exactly the examples as they are presented, and the negative variants are obtained from the positive ones by adding a second constructor $\mathsf{d} : \iota_0$. In the positive variants, we obtain $|\iota_n| = 1$, whereas in the negative variants $|\iota_n|$ becomes huge. Note that the encoding size of the TRSs in Example 26 grows linearly in n, whereas it is quadratic in Example 27.

The third set of example TRSs are encodings of the pigeon-hole principle, cf. Example 49, where the encoding of the TRSs grows cubic in n. The positive examples are $\mathcal{R}_p^{n,n}$ and the negative examples are $\mathcal{R}_p^{n+1,n}$.

Example 49 (Pigeon-Hole TRSs $\mathcal{R}_p^{n,m}$)**.** The signature of $\mathcal{R}_p^{n,m}$ consists of a single sort ι which consists of n constructors. All constructors are constants, e.g., $C = \{\mathsf{c}_1 : \iota, \ldots, \mathsf{c}_n : \iota\}$. Moreover, there is a single defined function symbol $\mathsf{f} : \iota^{m+1} \to \iota$. The TRS has quadratically many left-hand sides, namely all terms of the shape

$$\mathsf{f}(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_{j-1}, y, x_{j+1}, \ldots, x_{m+1}) \text{ where } 1 \le i < j \le m+1.$$

This TRS is pattern complete iff $n \le m$.

We perform the experiments by running both algorithms on these examples with increasing n, until either $n = 100$ is reached, or until some TRS $\mathcal{R}_n$ is reached such that pattern completeness of $\mathcal{R}_n$ cannot be decided within a 60 s timeout.[6]

The overall interpretations of Table 1 seems easy: the new algorithm of Section 5 clearly outperforms the FSCD algorithm of Section 4. Still, we like to give some remarks.

[5] 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 s when turning off the pattern completeness check.

[6] We actually invoke each algorithm on each TRS $\mathcal{R}_n$ for three times, and then measure whether the average time among the three execution times is above 60 s.

Table 1
Experimental results for the non-linear example TRSs $\mathcal{R}_n$. The numbers indicate the maximum number n between 1 and 100, such that pattern completeness of $\mathcal{R}_n$ was successfully determined within a 60 s timeout.

	FSCD		NEW	
	pos. variant	neg. variant	pos. variant	neg. variant
Example 26	12	12	100	100
Example 27	100	11	100	100
Example 49	8	100	11	100

- In the positive variant of Example 27, even the old algorithm could quickly solve all 100 input TRSs. This is in contrast to Example 41 where an exponential execution was displayed. The experiments reveal that the exponential behavior is just a worst-case scenario, and that actual implementations may find other executions with the FSCD algorithm that are quickly able to prove pattern completeness of these TRSs.
- As indicated in Example 46, deciding pattern completeness of Examples 26 and 27 becomes trivial due to the optimized rules for treating small and large cardinalities. Note that without the bounded computation of cardinalities, the negative examples variants cause a timeout for $n \geq 34$ on these examples, cf. Section 5.6.
- The pigeon hole TRSs in Example 49 are of shape where the transition to finite variable form is quickly performed in both algorithms. The negative variant can immediately be solved using (card-large), but also the implementation of the FSCD algorithm quickly finds a counterexample to pattern completeness. For the positive variants $\mathcal{R}_p^{n,n}$, both algorithms basically perform a full enumeration of all n^n many tuples as arguments of function f with a little bit of pruning. Here, the dedicated solver for problems in finite variable form working on integers (cf. Section 6.5) is a bit faster than the FSCD algorithm which does an enumeration of terms, but both algorithms suffer from the exhaustive enumeration.

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

8. 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. Moreover, the implementation has optimal asymptotic complexity: the algorithm has non-deterministic polynomial time complexity for returning negative answers, and the decision problem in co-NP hard.

We see some opportunities for future work. First, one can integrate an improved strategy to select variables for (instantiate) and (instantiate"), in particular since permutations in the input cause severe differences in runtime. One can also try to further improve the implementation, e.g., by following suggestions of Sestoft [14, 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 counterexample 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 constructors such as associativity and commutativity.

CRediT authorship contribution statement

René Thiemann: Writing – review & editing, Writing – original draft, Software, Formal analysis, Conceptualization; **Akihisa Yamada:** Writing – review & editing, Writing – original draft, Software, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

René Thiemann reports financial support was provided by Austrian Science Fund. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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 of this article and of the previous FSCD paper for their helpful remarks and suggestions.

This research was funded in part by the Austrian Science Fund (FWF) [Grant 10.55776/I5943]. For open access purposes, the author has applied a CC BY public copyright license to any author accepted manuscript version arising from this submission.

References

[1] R. Thiemann, A. Yamada, A verified algorithm for deciding pattern completeness, in: J. Rehof (Ed.), 9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10–13, 2024, Tallinn, Estonia, 299 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 27:1–27:17. https://doi.org/10.4230/LIPICS.FSCD.2024.27

[2] D. Kapur, P. Narendran, H. Zhang, On sufficient-completeness and related properties of term rewriting systems, Acta Inform. 24 (4) (1987) 395–415. https://doi.org/10.1007/BF00292110

[3] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL - A Proof Assistant for Higher-Order Logic, 2283 of *Lecture Notes in Computer Science*, Springer, 2002. https://doi.org/10.1007/3-540-45949-9

[4] T. Aoto, Y. Toyama, Ground confluence prover based on rewriting induction, in: D. Kesner, B. Pientka (Eds.), 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22–26, 2016, Porto, Portugal, 52 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 33:1–33:12. https://doi.org/10.4230/LIPICS.FSCD.2016.33

[5] U.S. Reddy, Term rewriting induction, in: M.E. Stickel (Ed.), 10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24–27, 1990, Proceedings, 449 of *Lecture Notes in Computer Science*, Springer, 1990, pp. 162–177. https://doi.org/10.1007/3-540-52885-7_86

[6] A. Lazrek, P. Lescanne, J. Thiel, Tools for proving inductive equalities, relative completeness, and omega-completeness, Inf. Comput. 84 (1) (1990) 47–70. https://doi.org/10.1016/0890-5401(90)90033-E

[7] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata: techniques and applications, 2007. https://jacquema.gitlabpages.inria.fr/files/tata.pdf.

[8] S. Falke, D. Kapur, Rewriting induction + linear arithmetic = decision procedure, in: B. Gramlich, D. Miller, U. Sattler (Eds.), Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26–29, 2012. Proceedings, 7364 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 241–255. https://doi.org/10.1007/978-3-642-31365-3_20

[9] C. Kop, Quasi-reductivity of logically constrained term rewriting systems, CoRR (2017). http://arxiv.org/abs/1702.02397.

[10] A. Bouhoula, F. Jacquemard, Sufficient completeness verification for conditional and constrained TRS, J. Appl. Log. 10 (1) (2012) 127–143. https://doi.org/10.1016/j.jal.2011.09.001

[11] A. Bouhoula, Simultaneous checking of completeness and ground confluence for algebraic specifications, ACM Trans. Comput. Log. 10 (3) (2009) 20:1–20:33. https://doi.org/10.1145/1507244.1507250

[12] A. Krauss, Partial and nested recursive function definitions in higher-order logic, J. Autom. Reasoning 44 (4) (2010) 303–336. https://doi.org/10.1007/s10817-009-9157-2

[13] A. Laville, Lazy pattern matching in the ML language, in: K.V. Nori (Ed.), Foundations of Software Technology and Theoretical Computer Science, Seventh Conference, Pune, India, December 17–19, 1987, Proceedings, 287 of *Lecture Notes in Computer Science*, Springer, 1987, pp. 400–419. https://doi.org/10.1007/3-540-18625-5_64

[14] P. Sestoft, MK pattern match compilation and partial evaluation, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12–16, 1996, Selected Papers, 1110 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 446–464. https://doi.org/10.1007/3-540-61580-6_22

[15] H. Comon, Sufficient completness, term rewriting systems and "anti-unification", in: J.H. Siekmann (Ed.), 8th International Conference on Automated Deduction, Oxford, England, July 27, - August 1, 1986, Proceedings, 230 of *Lecture Notes in Computer Science*, Springer, 1986, pp. 128–140. https://doi.org/10.1007/3-540-16780-3_85

[16] J. Thiel, Stop losing sleep over incomplete data type specifications, in: K. Kennedy, M.S.V. Deusen, L. Landweber (Eds.), Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984, ACM Press, 1984, pp. 76–82. https://doi.org/10.1145/800017.800518

[17] D. Kapur, P. Narendran, D.J. Rosenkrantz, H. Zhang, Sufficient-completeness, ground-reducibility and their complexity, Acta Inform. 28 (4) (1991) 311–350. https://doi.org/10.1007/BF01893885

[18] A. Yamada, R. Thiemann, Sorted terms, Arch. Form. Proofs (2024). https://isa-afp.org/entries/Sorted_Terms.html, Formal proof development.

[19] R. Thiemann, A. Yamada, Verifying a decision procedure for pattern completeness, Arch. Form. Proofs (2024). https://isa-afp.org/entries/Pattern_Completeness.html, Formal proof development.

[20] C. Sternagel, R. Thiemann, First-order terms, Arch. Form. Proofs (2018). https://isa-afp.org/entries/First_Order_Terms.html, Formal proof development.

[21] A. Middeldorp, A. Lochmann, F. Mitterwallner, First-order theory of rewriting for linear variable-separated rewrite systems: automation, formalization, certification, J. Autom. Reason. 67 (2) (2023) 14. https://doi.org/10.1007/S10817-023-09661-7

[22] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, M. Suda, SAT Competition 2020, Artif. Intell. 301 (2021) 103572. https://doi.org/10.1016/J.ARTINT.2021.103572