A verified algorithm for deciding pattern completeness and related properties*

René Thiemann

Universität Innsbruck, Austria
rene.thiemann@uibk.ac.at

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 an algorithm that decides pattern completeness and that can be used to ensure quasi-reducibility. One of the advantages of the algorithm is its simple structure: it is similar to that of a regular matching algorithm, and it avoids the enumeration of all terms up to a given depth (the latter is required in an existing decision procedure for quasi-reducibility.) Despite having a simple structure, termination and soundness proofs for the algorithm are not immediate. However, these properties have been verified in Isabelle/HOL.

1 Introduction

Consider programs written in a declarative style such as functional programs or term rewrite systems, where evaluation is defined by pattern matching. In several applications it is important to know that evaluation of a given program cannot get stuck, i.e., the programs should be sufficiently complete. For instance in Isabelle/HOL [7], a function definition must be sufficiently complete since HOL is a logic of total functions. And methods that are based on rewriting induction [1, 8] require similar completeness results, e.g., for proving ground confluence.

In both applications the evaluation mechanism can be described as a set of rules $\ell \to r$ where evaluation replaces instances of left-hand sides (lhss) $\ell\sigma$ by instances of right-hand sides $r\sigma$. Let L be the set of lhss of some set of rules. We consider programs where lhss are first-order terms over some finite signature $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ where $\mathcal{C}$ are constructor symbols and $\mathcal{D}$ are defined symbols. Hence, input values to a function are represented by constructor ground terms, denoted by $\mathcal{T}(\mathcal{C})$. We further assume a typed setting where we consider first-order monomorphic types, i.e, every function symbol of arity n has a type $\tau_1 \times \cdots \times \tau_n \to \tau_0$, where each type τ_i is just a name and τ_0 is called the target type. Also variables are typed and we write $\mathcal{V}_\tau$ for the set of variables of type τ. We define $\mathcal{T}(\mathcal{C})_\tau$ as the set of constructor ground terms that have type τ, and we assume $\mathcal{T}(\mathcal{C})_\tau \neq \emptyset$ for all types τ.

We can now define a first notion to describe that a program cannot get stuck.

Definition 1 (Pattern Completeness of Programs)**.** *A program with lhss L is pattern complete, if for all terms $f(t_1, \ldots, t_n)$, with $f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathcal{D}$ and $(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{C})_{\tau_1} \times \cdots \times \mathcal{T}(\mathcal{C})_{\tau_n}$, there is some $\ell \in L$ such that $t = f(t_1, \ldots, t_n)$ is matched by ℓ.*

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

Example 2. *Let* $\mathcal{C}_{\mathbb{N}} = \{\mathsf{true} : \mathbb{B}, \mathsf{false} : \mathbb{B}, \mathsf{0} : \mathbb{N}, \mathsf{s} : \mathbb{N} \to \mathbb{N}\}$ *be the set of constructors to represent the Booleans and natural numbers in Peano notation. We consider a program* $\mathcal{R}_{\mathbb{N}}$ *that defines a function to compute whether a natural number is even, i.e.,* $\mathcal{D} = \{\mathsf{even} : \mathbb{N} \to \mathbb{B}\}$.

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

This program is pattern complete, since no matter which number n *we provide as argument, one of the 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 types: without them $\mathsf{even}(\mathsf{s}(\mathsf{true}))$ *would contradict completeness.*

An alternative notion to pattern completeness is quasi-reducibility [5] where the difference is that the matching can happen for an arbitrary subterm.

Definition 3 (Quasi-Reducibility of Programs)**.** *A program with lhss* L *is quasi-reducible, if all terms* $f(t_1, \ldots, t_n)$*, with* $f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathcal{D}$ *and* $(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{C})_{\tau_1} \times \cdots \times \mathcal{T}(\mathcal{C})_{\tau_n}$*, there is some* $\ell \in L$ *such that a subterm of* $t = f(t_1, \ldots, t_n)$ *is matched by* ℓ*.*

Clearly, pattern completeness implies quasi-reducibility, and if the root symbols of all lhss in a program are within $\mathcal{D}$, then the two notions coincide. The following example illustrates the difference between the two notions.

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

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

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

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

But the program is not pattern complete since $\mathsf{even}(\mathsf{s}(\mathsf{p}(\mathsf{0})))$ *is not matched by any lhs.*

Kapur et al. proved decidability of quasi-reducibility [5]. They show that one may replace the quantification over all constructor ground terms $t_1, \ldots, t_n$ in Definition 3 by a bounded quantification where the depth of the terms t_i is restricted by d, a number that can computed from L; overall their decision procedure requires to enumerate exponentially many terms whenever $\mathcal{C}$ contains a symbol of arity 2 or larger. We are aware of two other algorithms to deduce quasi-reducibility in more complex settings, e.g., where rules may be constrained by arithmetic constraints such as "this rule is only applicable if $x > 0$" [4, 6], but both algorithms do not properly generalize the result of Kapur et al. since they are restricted to linear lhss. Bouhoula and Jacquemard [3] also designed an algorithm in a more complex setting with conditions and constraints, and a back-end that is based on constrained tree automata techniques. Since their soundness result is restricted to ground confluent systems, their algorithm is not applicable in our use-case, since we want to verify ground confluence proofs on methods that rely upon quasi-reducibility. Finally, Bouhoula developed an algorithm to verify ground confluence and sufficient completeness at the same time [2], where we are not sure whether it can also be used to just ensure completeness, e.g., for non-ground confluent systems.

In this paper we will provide a simple algorithm to decide pattern completeness. It avoids to always enumerate all terms up to given depth, it is not restricted to linear lhss, and it does

not require tree automata algorithms. It can also be used as a sufficient criterion for quasi-reducibility, and as a decision procedure for those programs where all lhss have a defined symbol as root.

2 Pattern Completeness – The Linear Case

Before we design the new decision procedure for pattern completeness we first reformulate and generalize this notion. A slightly more general notion than pattern completeness is already provided by Aoto and Toyama [1]. They define the concept of a *cover*, where L *covers* a term t if for all σ_{cg} there is some $\ell \in L$ that matches $t\sigma_{cg}$ (here, σ_{cg} represents some arbitrary constructor ground substitution where all variables are replaced by constructor ground terms). Hence, pattern completeness can be formulated as the question whether $f(x_1, \ldots, x_n)$ is covered by L for all defined symbols f where the x_i's are distinct variables.

We generalize the notion of a cover further into a pattern problem.

Definition 5 (Matching Problem and Pattern Problem)**.** *A matching problem is a finite set $mp = \{(t_1, \ell_1), \ldots, (t_n, \ell_n)\}$ that contains arbitrary pairs of terms. A pattern problem is a finite set $pp = \{mp_1, \ldots, mp_k\}$ of matching problems.*

A matching problem mp is solvable w.r.t. some constructor ground substitution σ_{cg} if there is some substitution γ such that $t_i\sigma_{cg} = \ell_i\gamma$ for all $(t_i, \ell_i) \in mp$. A pattern problem pp is solvable if for each constructor ground substitution σ_{cg} there is some $mp \in pp$ such that mp is solvable w.r.t. σ_{cg}. A set of pattern problems P is solvable if each $pp \in P$ is solvable.

We further introduce a special matching problem $\bot_{mp}$ that represents an unsolvable matching problem. Similarly, we define $\top_{pp}$ as a new pattern problem that is always solvable. Finally, $\bot_P$ represents a new unsolvable set of pattern problems.

Hence, the question of whether L covers t can be encoded in the pattern problem $\{\{(t, \ell)\} \mid \ell \in L\}$. Similarly, Aoto and Toyama's notion of strong quasi-reducibility [1] can also be encoded as a pattern problem: $\bigcup_{t \in \{x_1,\ldots,x_n, f(x_1,\ldots,x_n)\}} \{\{(t, \ell)\} \mid \ell \in L\}$ expresses that one tries to find a match at the root ($t = f(x_1, \ldots, x_n)$) or a match for a direct subterm ($t = x_i$). Finally, the question of whether a program with lhss L and defined symbols $\mathcal{D}$ is pattern complete w.r.t. Definition 1 is expressible as solvability of the set of pattern problems $\{\{\{(f(x_1, \ldots, x_{n_f}), \ell)\} \mid \ell \in L\} \mid f \in \mathcal{D}\}$ where n_f is the arity of f and the variables $x_1, \ldots, x_{n_f}$ are distinct.

The following inference rules describe a decision procedure to determine solvability of *linear* pattern problems. A matching problem $\{(t_1, \ell_1), \ldots, (t_n, \ell_n)\}$ is linear if each ℓ_i is linear and the variables of ℓ_i and ℓ_j are disjoint for $i \neq j$. A pattern problem is linear if all its matching problems are linear.

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

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

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

$$\{(t, x)\} \uplus mp \to mp \qquad \text{(match)}$$

On top of this we define simplification rules $\Rightarrow$ for pattern problems.

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

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

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

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

$$\{pp\} \uplus P \Rrightarrow \{pp'\} \cup P \qquad \text{if } pp \Rightarrow pp' \qquad \text{(simp-pp)}$$
$$\{\emptyset\} \uplus P \Rrightarrow \bot_P \qquad \text{(failure)}$$
$$\{\top_{pp}\} \uplus P \Rrightarrow P \qquad \text{(remove-pp)}$$
$$\{pp\} \uplus P \Rrightarrow \{pp\sigma_{x,c} \mid c \in \mathcal{C}_\tau\} \cup P \qquad \text{if } mp \in pp,\ (x, f(\ldots)) \in mp,\ \text{and } x \in \mathcal{V}_\tau \qquad \text{(instantiate)}$$

Here, $\mathcal{C}_\tau$ is the set of constructors with target type τ and $\sigma_{x,c}$ is a substitution which just replaces x by $c(x_1, \ldots, x_n)$ where n is the arity of c and $x_1, \ldots, x_n$ are fresh and distinct variables. The pattern problem $pp\sigma_{x,c}$ is obtained from pp by changing every pair (t, ℓ) in every matching problem of pp to $(t\sigma_{x,c}, \ell)$.

Clearly, (decompose) and (clash) correspond to a standard matching algorithm. Similarly, (match) is standard for matching with linear lhss, but will cause problems in the non-linear case. Nearly all of the other rules mainly correspond to the universal and existential quantification that is done in the definition of solvability. The only exception is (instantiate). Here a matching algorithm would detect a failure since a variable x is never matched by a non-variable term $f(\ldots)$. However, since the x in our setting just represents an arbitrary constructor ground term, we need to make a case analysis on the outermost constructor. This is done by replacing $x \in \mathcal{V}_\tau$ by all possible constructor ground terms of shape $c(x_1, \ldots, x_n)$ for all $c \in \mathcal{C}_\tau$.

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

Theorem 7 (Decision Procedure for Solvability of Linear Pattern Problems)**.**

- *$\Rrightarrow$ is terminating.*
- *Whenever $P \Rrightarrow^! P'$ then $P' \in \{\emptyset, \bot_P\}$.*
- *Whenever P is linear and $P \Rrightarrow P'$ then P' is linear, and P is solvable iff P' is solvable.*
- *Whenever P is linear then P is solvable iff $P \Rrightarrow^! \emptyset$.*

So, solvability of linear pattern problems is decidable. Regarding the complexity, one can prove an exponential upper bound on the number of $\Rrightarrow$-steps. However, there might be room for improvement: for all examples we considered so far, there also is a strategy such that only polynomially many $\Rrightarrow$-steps are required, e.g., by changing the order of variables on which (instantiate) is applied.

Example 8. *The algorithm validates that $\mathcal{R}_\mathbb{N}$ in Example 2 is pattern complete. In the execution of the algorithm we interpret sets as multisets.*

$$\begin{aligned}
P = \{\ & \{\{(\mathsf{even}(y), \mathsf{even}(0))\}, \{(\mathsf{even}(y), \mathsf{even}(\mathsf{s}(0)))\}, \{(\mathsf{even}(y), \mathsf{even}(\mathsf{s}(\mathsf{s}(x))))\}\}\ \} \\
\Rrightarrow^3\ & \{\ \{\{(y, 0)\}, \{(y, \mathsf{s}(0))\}, \{(y, \mathsf{s}(\mathsf{s}(x)))\}\}\ \} \\
\Rrightarrow\ & \{\ \{\{(0,0)\}, \{(0, \mathsf{s}(0))\}, \{(0, \mathsf{s}(\mathsf{s}(x)))\}\},\ \{\{(\mathsf{s}(z), 0)\}, \{(\mathsf{s}(z), \mathsf{s}(0))\}, \{(\mathsf{s}(z), \mathsf{s}(\mathsf{s}(x)))\}\}\ \} \\
\Rrightarrow^6\ & \{\ \{\emptyset, \bot_{mp}, \bot_{mp}\},\ \{\bot_{mp}, \{(z, 0)\}, \{(z, \mathsf{s}(x))\}\}\ \} \\
\Rrightarrow^3\ & \{\ \{\{(z, 0)\}, \{(z, \mathsf{s}(x))\}\}\ \} \\
\Rrightarrow\ & \{\ \{\{(0,0)\}, \{(0, \mathsf{s}(x))\}\},\ \{\{(\mathsf{s}(y), 0)\}, \{(\mathsf{s}(y), \mathsf{s}(x))\}\}\ \} \\
\Rrightarrow^5\ & \{\ \{\emptyset, \bot_{mp}\},\ \{\bot_{mp}, \emptyset\}\ \} \\
\Rrightarrow^4\ & \emptyset
\end{aligned}$$

3 Pattern Completeness – The General Case

For achieving soundness for the non-linear case we have to modify the (match) rule.

Definition 9 (Match Rule for the General Case)**.**

$$\{(t,x)\} \uplus mp \to mp \qquad \textit{if for all } (t',\ell) \in mp,\ x \textit{ does not occur } \ell \qquad \text{(match')}$$

In the linear case the additional occurrence check in rule (match') is always satisfied, so we can still simulate the algorithm for the linear case with this modified rule. Soundness and termination of $\Rrightarrow$ still are satisfied, even for non-linear inputs.

However, after the switch from rule (match) to (match') it can happen that $\Rrightarrow$ gets stuck, e.g., if there is a matching problem $\{(t,x),(t',x)\}$ for $t \neq t'$. To treat these cases we have to add further simplification rules. In order to do so, we need to distinguish between finite and infinite types τ, i.e., whether the set $\mathcal{T}(\mathcal{C})_\tau$ is finite or infinite. To illustrate the problem, consider a program with three left-hand sides: $f(x,x,y)$, $f(x,y,x)$, and $f(y,x,x)$. If x is a variable of a finite type that just allows two different values, such as the Booleans, then these left-hand sides cover all cases. If the type has infinitely many values, such as lists, then the left-hand sides do not suffice, indicating an unsolvable problem. So, we must be able to instantiate in the finite-type case. However, we cannot allow an instantiation in the infinite-type case, since otherwise the resulting inference rules would no longer be terminating.

As final preparation for the new inference rules we define (the only two) reasons on why two terms differ. We say that terms $t \neq t'$ clash if $t|_p = f(\dots) \neq g(\dots) = t'|_p$ with $f \neq g$ for some shared position p of t and t'. The terms $t \neq t'$ differ in variable y if $t|_p \neq t'|_p$ and $y \in \{t|_p, t'|_p\}$ for some shared position p.

Definition 10 (Inference Rules for General Pattern Problems)**.** *We take all rules of the linear algorithm with the following modifications.*

- *Rule (match) is replaced by (match').*
- *We add the following three rules to avoid to get stuck.*

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

$$\{pp\} \uplus P \Rrightarrow \{pp\sigma_{x,c} \mid c \in \mathcal{C}_\tau\} \cup P \qquad \text{(instantiate')}$$
$$\textit{if } mp \in pp,\ \{(t,y),(t',y)\} \subseteq mp,\ t \textit{ and } t' \textit{ differ in variable } x \in \mathcal{V}_\tau, \textit{ and } \tau \textit{ is finite}$$

$$\{pp\} \uplus P \Rrightarrow \bot_P \qquad \textit{if for each } mp \in pp \textit{ there are } \{(t,y),(t',y)\} \subseteq mp \qquad \text{(failure')}$$
$$\textit{such that } t \textit{ and } t' \textit{ differ in variable } x \in \mathcal{V}_\tau \textit{ and } \tau \textit{ is infinite}$$

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

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

Overall, we arrive at a similar theorem to the linear case, though its proof is much more evolved. It has been proven in Isabelle (2700 lines), based on IsaFoR[1] and on a library on sorted terms by Akihisa Yamada.

[1] http://cl-informatik.uibk.ac.at/isafor/

Theorem 11 (Decision Procedure for Solvability of Pattern Problems)**.**

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

The formalization in Isabelle also contains a verified list-based implementation of the abstract inference rules. It fixes a strategy where first $\Rightarrow$-steps are applied exhaustively. Rules (instantiate) and in particular (instantiate') get applied as late as possible. However, this implementation is not fully working, as it expects a function to compute whether a given type τ is infinite or not, and we did not yet verify a suitable algorithm for this subtask.

Note that it is possible to extend $\Rrightarrow$ in a way that it provides a witness constructor ground substitution in case a pattern problem is not solvable. To this end one has to store the substitutions that have been applied via the two rules for instantiation; and in case rule (failure') has been used, a final constructor-ground substitution can be generated by following the construction in the soundness proof of that rule.

It remains open whether a similar syntax directed decision procedure for quasi-reducibility can be designed, i.e., without an explicit enumeration of terms.

Acknowledgements We thank the anonymous reviewers for their helpful remarks and for their references to related work.

References

[1] Takahito Aoto and Yoshihito Toyama. Ground confluence prover based on rewriting induction. In *Proc. FSCD 2016*, volume 52 of *LIPIcs*, pages 33:1–33:12, 2016.

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

[3] Adel Bouhoula and Florent Jacquemard. Sufficient completeness verification for conditional and constrained TRS. *J. Appl. Log.*, 10(1):127–143, 2012.

[4] Stephan Falke and Deepak Kapur. Rewriting induction + linear arithmetic = decision procedure. In *Proc. IJCAR 2012*, volume 7364 of *LNCS*, pages 241–255, 2012.

[5] Deepak Kapur, Paliath Narendran, and Hantao Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24(4):395–415, 1987.

[6] Cynthia Kop. Quasi-reductivity of logically constrained term rewriting systems. *CoRR*, abs/1702.02397, 2017.

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[8] Uday S. Reddy. Term rewriting induction. In *Proc. CADE 1990*, volume 449 of *LNCS*, pages 162–177, 1990.