# Literature Study on Algorithms for Pattern Completeness

Vrije Universiteit Amsterdam

Daniel Köves
d.koves@student.vu.nl

Supervisors
Femke van Raamsdonk
Jörg Endrullis

November 2024

# 1 Introduction

The following literature review aims to explore pattern completeness and the related notion of quasi-reducibility of term rewriting systems. Pattern completeness denotes the property that given a term rewrite system with left hand sides $L$, for any given term $f(t)$ where $f$ is a defined symbol, it is matched by some left hand side $\ell \in L$. Quasi-reducibility relaxed this definition, allowing for matches to happen under the root, i.e. given any term $t = f(x)$, a left hand side $\ell \in L$ matches a subterm of $t$. Pattern completeness intuitively means that the definition of $f$ is *complete*, e.g. when we think about functional programs that use pattern matching. Whereas with quasi-reduciblity we mean that the term can further be reduced in the system, or the execution of a program doesn't get stuck.

The main entry-point of the review is the paper by Thiemann and Yamada [1], in which the authors present a novel algorithm to decide pattern completeness. The algorithm, described in detail in section 3, is compared against other implementations, namely the *complement algorithm* of Lazrek et al. [2]. Moreover, tree automata-based solution proved useful at determining pattern completeness, therefore a short introduction at that construction is also detailed. Furthermore, other notable works of literature are also briefly presented.

The organisation of the paper is the following: the rest of section 1 introduces some mathematical preliminaries and details the problem at hand – pattern completeness and quasi-reducibility of term rewrite systems. The algorithm in the Thiemann and Yamada paper is explored in detail and analysed in section 3, whereas the complement algorithm is presented in the sections following it. The paper also briefly mentions further notable related work, such as quasi-reducibility and decidability thereof as presented by Kapur et al [3], further techniques for pattern matching and pattern completeness in section 6. Following the review, the main algorithms are compared and their differences are discussed in section 7, and finally, concluding remarks are found in section 8.

The main focus of the review is to discuss algorithms for deciding pattern completeness. The notion of pattern completeness comes up in functional programming, namely, most languages that work by means of pattern matching require that the defined patterns are complete (and warn otherwise). Running a program with an incomplete pattern would result in untimely termination of the program with an exception. These runtime errors are the "worst" type of errors a program can encounter, therefore, it is crucial to ensure it does not happen. Figure 1 shows such an example, the case matching `Nothing` is missing.

```
maybeAddOne :: Maybe Int -> Int
maybeAddOne (Just n) = n + 1
```

Figure 1: Haskell snippet with an incomplete pattern

A notion related to pattern completeness is quasi-reducibility [3], that ensures that the evaluation cannot get stuck. The difference between the two

notions are discussed in section 2.2.

# 2 Preliminaries

Given a signature $\Sigma$ a set of function symbols $f \in \Sigma$ and a set of variables $\mathcal{X}$, we say that terms $\mathcal{T}(\Sigma, \mathcal{X})$ are the smallest set such that $\mathcal{X} \subseteq \mathcal{T}(\Sigma, \mathcal{X})$ and $f(t_1, ..., t_n) \in \mathcal{T}(\Sigma, \mathcal{X})$ given $f \in \Sigma$ of arity $n$ and variables $t_1, ..., t_n \in \mathcal{X}$. By $\mathcal{V}ar(t)$ we denote the set of variables in $t$. Ground terms, denoted as $T(\Sigma)$ are terms without variables. A term $s$ is subterm of $t$ if $s = t$ or $t = f(t_1, ..., t_n)$ and s is subterm of some $t_i$ given $1 \leq i \leq n$. A set of positions $\mathcal{P}os(t)$ of term $t$ is defined as the set $\{\epsilon\}$ when $t \in \mathcal{X}$, otherwise the set $\{\epsilon\} \cup \{ip \mid p \in \mathcal{P}os(t_i)\}$. By $t|_p$ we note the subterm of $t$ at position $p$.

Given a set of sorts $\mathcal{S}$, a sorted set $\mathcal{V}$ is a set in which each element is associated with a sort $\iota \in \mathcal{S}$ written as $v : \iota \in \mathcal{V}$. Given a sorted signature $\mathcal{F}$ and sorted set of variables $\mathcal{V}$, we define sorted terms as $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

A substitution $\sigma$ is a mapping $\mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$. Notations include $\sigma(t)$, $t\sigma$ or $t^\sigma$, all meaning the substitution $\sigma$ applied to term $t$.

A rewrite rule $(\ell \to r)$ is a pair $(\ell, r)$ such that $\ell \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell)$.

A term rewrite system $\mathcal{R}$ is a pair $(\Sigma, R)$ where $R$ is a set of rewrite rules between $\mathcal{T}(\Sigma, \mathcal{X})$. In a TRS $\mathcal{R}$ over $\Sigma$, defined symbols $\mathcal{D}$ are the root function symbols of the left hand sides of the rewrite rules, and constructor terms $\mathcal{C}$ are defined as $\Sigma \setminus \mathcal{D}$. Inputs to functions are therefore represented by constructor ground terms.

## 2.1 Pattern Completeness and the Pattern Problem

The *matching problem* asks, given two terms $s$ and $t$, whether there exists a substitution $\sigma$ such that $s\sigma = t$. The direction is from $s$ to $t$, namely, we try to match $s$ to $t$ by $\sigma$. A matching problem $mp$ is represented as a set $\{(t_1, \ell_1), ..., (t_n, \ell_n)\} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$, assuming $\mathcal{V}$ and $\mathcal{X}$ do not overlap. A pattern problem $pp$ is a finite set of matching problems. A matching problem is complete, if given a constructor ground substitution $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{C})$, there is a substitution $\gamma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $t\sigma = \ell\gamma$ for all $(t, \ell) \in mp$. A pattern problem is complete if for each constructor ground substitution $\sigma$ there is some $mp \in pp$ that is complete. A set $P$ of pattern problems is complete if all $pp \in P$ are complete.

A program with left-hand sides $L$ is pattern complete, if every basic ground term – which are defined as terms of the form $f(t_1, ..., t_n)$ with $f \in \mathcal{D}$ and $\{t_1, ..., t_n\} \subseteq \mathcal{T}(\mathcal{C}, \mathcal{X})$ –, is matched by some $\ell \in L$. The question whether a program with left hand sides $L$ and defined symbols $\mathcal{D}$ is pattern complete can be expressed with the following set of pattern problems[1]:

$$P = \{\{\{(f(x_1, ..., x_n), \ell)\} \mid \ell \in L\} \mid f : \iota_1, ..., \iota_n \to \iota_0 \in \mathcal{D}\}$$

**Example 2.1.** Consider as example the following sorted term rewrite system to calculate whether a natural number is even. The TRS $\mathcal{R}_{\mathbb{N}}$ is given as, adapted from Example 1 in [1]:

$$\mathcal{C}_{\mathbb{N}} = \{\text{true} : \mathbb{B}, \text{false} : \mathbb{B}, 0 : \mathbb{N}, s : \mathbb{N} \to \mathbb{N}\}$$
$$\mathcal{D}_{\mathbb{N}} = \{\text{even} : \mathbb{N} \to \mathbb{B}\}$$

with the following rules:

$$\text{even}(0) \to \text{true} \qquad \text{even}(s(0)) \to \text{false} \qquad \text{even}(s(s(x))) \to \text{even}(x)$$

In this setting consider the following matching problems:

$$mp_1 = \{(z, 0)\} \qquad mp_2 = \{(\text{even}(z), 0)\}$$

Matching problem $mp_1$ is complete with respect to $\sigma = \{z \mapsto 0\}$, however $mp_2$ is incomplete since there exists no $\sigma$ such that $\text{even}(x)^\sigma = 0$. The set of pattern problems describing this program would be, where $z$ stands for some constructor ground term:

$$P = \{\{\{(\text{even}(z), \text{even}(0))\}, \{(\text{even}(z), \text{even}(s(x)))\}, \{(\text{even}(z), \text{even}(s(s(x))))\}\}\}$$

## 2.2 Quasi-reducibility

A notion related to pattern completeness is quasi-reducibility, the difference being that as per the definition in Section 2.1, pattern completeness does not allow for matching below the root term. This condition is relaxed when talking about quasi-reducibility. The notion of quasi-reducibility aligns with *sufficient* or *relative completeness* in [2, 4]. Take as example the extended version of Example 2.1 for integers using successor-predecessor notation, taken from Example 4 of the Thiemann and Yamada paper [1].

**Example 2.2.** Given TRS $\mathcal{R}_{\mathbb{Z}}$ using

$$\mathcal{C}_{\mathbb{Z}} = \{\text{true} : \mathbb{B}, \text{false} : \mathbb{B}, 0 : \mathbb{Z}, s : \mathbb{Z} \to \mathbb{N}, p : \mathbb{Z} \to \mathbb{N}\}$$
$$\mathcal{D}_{\mathbb{Z}} = \{\text{even} : \mathbb{Z} \to \mathbb{B}\}$$

with rules:

$$\begin{aligned} \text{even}(0) &\to \text{true} & \text{even}(s(0)) &\to \text{false} & \text{even}(s(s(x))) &\to \text{even}(x) \\ \text{even}(p(0)) &\to \text{false} & \text{even}(p(p(x))) &\to \text{even}(x) & & \\ s(p(x)) &\to x & p(s(x)) &\to x & & \end{aligned}$$

The term rewriting system is quasi-reducible, every term in the form $even(z)$ has a subterm that matches one of the rules. All integers are covered by the first rules and if our term contains both $s$ and $p$, the last two rules apply. However, the system is not pattern complete, because e.g. the term $\text{even}(s(p(0)))$ is not matched by any left hand side.

The difference between the notions of pattern completeness and quasi-reducibility is also illustrated by Example 3.5, in which it is algorithmically confirmed that $\mathcal{R}_{\mathbb{Z}}$ is not pattern complete, and in Example 4.3 in which it is algorithmically confirmed to be quasi-reducible.

# 3 Thiemann and Yamada's algorithm

The algorithm presented in the paper of Thiemann and Yamada [1] works on multisets of pattern problems and applies rules on the innermost matching problems, pattern problems and sets of pattern problems. Two iterations are presented, one dealing with only linear inputs (where no variable appears multiple times in the left-hand sides), and a further iteration with additional rules to handle non-linearity. The rules of the algorithm are presented here in a slightly different notation.

Matching problems (denoted as $mp$) are reduced using the following rules:

$$\begin{aligned}
\textbf{decompose} \quad & \{(f(t_1,...,t_n), f(l_1,...,l_n))\} \to \{(t_1,l_1),...,(t_n,l_n)\} \\
\textbf{match} \quad & \{(t,x)\} \in mp \to \varnothing \text{ if } \forall (t',l) \in mp.\ x \notin Var(l) \\
\textbf{clash} \quad & \{(f(...), g(...))\} \to \bot_{mp} \text{ if } f \neq g
\end{aligned}$$

For pattern problems (sets of matching problems – denoted as $pp$), the following rules apply:

$$\begin{aligned}
\textbf{remove-mp} \quad & \{\bot_{mp}\} \to \varnothing \\
\textbf{success} \quad & \{\varnothing\} \to \top_{pp}
\end{aligned}$$

Finally for sets of pattern problems (which is the input of the algorithm, denoted as $P$), the rules are as follows:

$$\begin{aligned}
\textbf{failure} \quad & \{\varnothing\} \to \bot_P \\
\textbf{remove-pp} \quad & \{\top_{pp}\} \to \varnothing \\
\textbf{instantiate} \quad & \{pp\} \to \text{Inst}(pp, x) \text{ if } \{(x, f(...))\} \in pp
\end{aligned}$$

The **match** rule removes a matching problem from the set in case variable $x$ does not occur in any other matching problems in the same set. This intuitively mean, we match variable $x$ to $t$.

The last rule, **instantiate**, is applicable in case variable $x$ is tried to be matched with some non-variable term $f(...)$. In this case, we construct a new pattern problem $pp\sigma_{x,c}$ for each constructor in $\mathcal{C}$, given as:

$$pp\sigma_{x,c} = \{\{(t\sigma_{x,c}, l) \mid (t,l) \in mp\} \mid mp \in pp\}$$

where

$$\sigma_{x,c} = [x \mapsto c(x_1,...,x_n)]$$

for each $c \in \mathcal{C}$ of arity $n$, and fresh variables $x_1,...,x_n$.

In order the deal with non-linearity, further rules are introduced so that the algorithm does not get stuck on non-linear input. These rules are as follows:

$$\begin{aligned}
\textbf{clash'} \quad & \{(t,x),(t',x)\} \in mp \to \bot_{mp} \text{ if } t \text{ and } t' \text{ clash} \\
\textbf{instantiate'} \quad & \{\{(t,x),(t',x)\}\} \in P \to \text{Inst}(pp, x) \\
\textbf{failure'} \quad & \{pp\} \in P \to \bot_P
\end{aligned}$$

In case of **instantiate'**, we can apply the rule if $t$ and $t'$ differ in variable $x$ of finite sort (such that $\{t \mid t : \iota \in \mathcal{T}(\mathcal{C})\}$ is a finite set).

In case of **failure'**, we need to fail the algorithm if within each $mp \in pp$ there exists $\{(t, x), (t', x)\}$ such that $t$ and $t'$ differ in variable $x$ of infinite sort. This special case is needed, as it would be impossible to instantiate a variable of infinite sort, if we find that terms differ in such variable. In case not all matching problems differ in such a variable, we can mark those problems locally as $\bot_{mp}$ (via **clash'**).

We say a term $t$ and $t'$ clash if $t|_p = f(...) \neq g(...) = t'|_p$. Terms $t$ and $t'$ differ in variable $x$ if $t|_p \neq t'|_p$ and $x \in \{t|_p, t'|_p\}$.

The order, in which the reduction steps are applied in the list-based implementation are given as follows:

1. Exhaustively apply **decompose**, **clash** and **clash'**
2. Exhaustively apply **match** and try to apply **failure'**
3. Invoke **instantiate** or **instantiate'** with preference for the former

## 3.1 Examples

The following examples illustrate certain executions of the algorithm. We assume the sort of natural numbers with one defined symbol $f : \mathbb{N} \to \mathbb{N}$, using the following constructors $\mathcal{C}_{\mathbb{N}} = \{0 : \mathbb{N}, s(x) : \mathbb{N} \to \mathbb{N}\}$.

**Example 3.1.** Linear case

Given a left-hand side of $\{f(0), f(s(x))\}$, the linear algorithm would compute:

$$
\begin{aligned}
P &= \{\{\{(f(a), f(0))\}, \{(f(a), f(s(x)))\}\}\} \\
\text{decompose} &\Rrightarrow^* \{\{\{(a, 0)\}, \{(a, s(x))\}\}\} \\
\text{instantiate} &\Rrightarrow^* \{\{\{(0, 0)\}\}, \{\{(s(z), s(x))\}\}\} \\
\text{match} &\Rrightarrow \{\{\bot_{mp}\}, \{\{(s(z), s(x))\}\}\} \\
\text{remove-mp} &\Rrightarrow^* \{\{\varnothing\}, \{\{(s(z), s(x))\}\}\} \\
\text{decompose} &\Rrightarrow \{\{\varnothing\}, \{\{(z, x)\}\}\} \\
\text{match} &\Rrightarrow \{\{\varnothing\}, \{\varnothing\}\} \\
\text{success} &\Rrightarrow^* \{\top_{pp}, \top_{pp}\} \\
\text{remove-pp} &\Rrightarrow^* \varnothing
\end{aligned}
$$

The algorithm concludes that the left hand sides are pattern complete.

**Example 3.2.** Linear case failure

The following example demonstrates how the algorithm would find an incomplete pattern:

$$\begin{aligned} P &= \{\{\{(f(a), f(s(x))\}\}\} \\ \text{decompose} &\Rrightarrow \{\{\{(a, s(x))\}\}\} \\ \text{instantiate} &\Rrightarrow \{\{\{(0, s(x))\}\}, \{\{(s(z), s(x))\}\}\} \\ \text{clash} &\Rrightarrow \{\{\bot_{mp}\}, \{(s(z), s(x))\}\} \\ \text{remove-mp} &\Rrightarrow \{\varnothing, \{(s(z), s(x))\}\} \\ \text{failure} &\Rrightarrow \bot_P \end{aligned}$$

Here we could have further reduced the pattern problem $\{\{(s(z), s(x))\}\}$, however, do the incomplete case $\{\{(0, s(x))\}\}$ resulting in $\varnothing$, the whole pattern problem reduces to $\bot_P$.

**Example 3.3.** General case

Given defined symbol $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ the following left-hand sides: $\{\{f(x, x)\}, \{f(x, y)\}\}$ the algorithm would compute:

$$\begin{aligned} P &= \{\{\{(f(a, b), f(x, x))\}, \{(f(a, b), f(x, y))\}\} \\ \text{decompose} &\Rrightarrow^* \{\{\{(a, x), (b, x)\}, \{(a, x), (b, y)\}\}\} \\ \text{clash'} &\Rrightarrow \{\{\bot_{mp}, \{(a, x), (b, y)\}\}\} \\ \text{remove-mp} &\Rrightarrow \{\{\{(a, x), (b, y)\}\}\} \\ \text{match} &\Rrightarrow^* \{\{\varnothing\}\} \\ \text{success} &\Rrightarrow^* \{\top_{pp}\} \\ \text{remove-pp} &\Rrightarrow^* \varnothing \end{aligned}$$

**Example 3.4.** Linear algorithm with non-linear input

The following example illustrates that the additional rules to match non-linear inputs are necessary to decide pattern completeness, using the non-linear left-hand side from Example 3.3:

$$\begin{aligned} P &= \{\{\{(f(a, b), f(x, x))\}\}\} \\ \text{decompose} &\Rrightarrow^* \{\{\{(a, x), (b, x)\}, \{(a, x), (b, y)\}\}\} \end{aligned}$$

At this stage, the only rule we might want to apply is **match**. However, due to the condition in that rule that the variable $x$ cannot appear in any right hand side within the matching problem, we fail to apply the rule. Therefore, the algorithm is stuck, indeed no other rule apply (not even **instantiate**, since $x$ is not a constructor term). The correct step in this case would be to apply **clash'**, available in the extension of the algorithm to the general case.

**Example 3.5.** Quasi-reducible LHS is not pattern complete

Let us apply the algorithm to Example 2.2. In that example the following term rewriting system $\mathcal{R}_{\mathbb{Z}}$ is presented, using

$$\mathcal{C}_{\mathbb{Z}} = \{\text{true} : \mathbb{B}, \text{false} : \mathbb{B}, 0 : \mathbb{Z}, s : \mathbb{Z} \to \mathbb{N}, p : \mathbb{Z} \to \mathbb{N}\}$$

with one defined symbol $\mathcal{D}_{\mathbb{Z}} = \{\text{even} : \mathbb{Z} \to \mathbb{B}\}$:

$$\begin{array}{ccc} \text{even}(0) \to \text{true} & \text{even}(s(0)) \to \text{false} & \text{even}(s(s(x))) \to \text{true} \\ \text{even}(p(0)) \to \text{false} & \text{even}(p(p(x))) \to \text{even}(x) & \\ s(p(x)) \to x & p(s(x)) \to x & \end{array}$$

The algorithm would compute:

$$\begin{aligned} P = &\{\{\{(even(z), even(0))\}, \{(even(z), even(s(0)))\}, \\ &\{(even(z), even(s(s(x))))\}, \{(even(z), even(p(0)))\}, \\ &\{(even(z), even(p(p(x))))\}, \{(even(z), even(p(p(0))))\}, \\ &\{(even(z), s(p(x)))\}, \{(even(z), p(s(x)))\}\} \\ \Rrightarrow^* &\{\{\{(s(z), 0)\}, \{(s(z), p(x))\}\}, \{\{(p(z), 0)\}, \{(p(z), s(x))\}\}\} \end{aligned}$$

Previous steps are omitted for brevity, the only problematic pattern problems are the ones above, all other cases are resolved with $\top_{pp}$. These cases, however, would result in clashes, therefore the whole pattern problem reduces to an empty set, marking $P$ as $\bot_P$.

## 3.2 Analysis

This sections details the correctness of the algorithm. Firstly, that the reduction steps have normal form $\{\varnothing, \bot_P\}$, secondly, that the algorithm terminates, by defining a measure of difference between pattern problems and showing that each step decreases in this order.

The normal forms of the reduction steps is one of $\{\varnothing, \bot_P\}$. The algorithm iteratively removes matching problems with **match** or marks them incomplete with **clash**. Using the definition of completeness in Section 2.1, a pattern problem $pp$ is complete if for each constructor ground substitution $\sigma$, there exists a matching problem $mp \in pp$ that is complete. Therefore, the pattern problem is marked $\top_{pp}$ by **success** when the pattern problem reduces to $\varnothing$ by removing all clashed and matching problems via **remove-mp**. When the pattern problem reduces to the empty set we can conclude that at least one **match** has taken place, therefore, the pattern is complete. On the other hand, when no match has taken place, the matching problem reduces to $\{\bot_{mp}\}$. The notation here is bit subtle, due to the nested empty sets. Namely, what leads to a pattern problem being marked successful is an empty pattern problem $\{\varnothing\}$. Furthermore, what leads to the whole execution marked as failure is a pattern problem containing only $\bot_{mp}$. The reader can refer to the examples in Section 3.1 for further clarification.

Furthermore, the algorithm is shown to be decidable and terminating [1]. The termination proof for the left-linear case is given by defining a measure of difference $|\ell - t|$ between terms $(\ell, t)$ of a matching problem as:

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

This measure is lifted to pattern problems by:

$$|pp|_{\text{diff}} = \sum_{mp \in pp, (t,\ell) \in mp} |\ell - t|$$

Then, the relation $\succ$ is defined for sets of pattern problems by extending $>$ to multisets $>^{\text{mul}}$ by:

$$P \succ P' \iff \{|pp|_{\text{diff}} \mid pp \in P\} >^{\text{mul}} \{|pp'|_{\text{diff}} \mid pp' \in P'\}$$

. The relation $\succ$ is strongly normalizing, each step of the algorithm weakly decreases, while the instantiate rule strictly decreases with respect to $\succ$ [1].

# 4 Complement algorithm

The *complement algorithm* presented by Lazrek, Lescanne, and Thiel in [2], represents a mechanism to conclude whether in a TRS $\mathcal{R}$, a defined symbol $f$ (called operator in the paper) is convertible to a set of constructors, denoted as *relative* completeness. The algorithm can also be used to decide quasi-reducibility and indirectly pattern completeness, as demonstrated by Example 4.3. The main idea behind the algorithm is to compute the *complement* of matched terms, then iteratively check whether the complement can be further reduced or matched. The complement of a term $t$ means the ground terms of $t$ and the complement of $t$ equal the set of constructor terms.

The algorithm works on pairs of sets $M_i$ and $N_i$, each iteration $M_{i+1}$ and $N_{i+1}$ are constructed from their previous counterpart. The algorithm starts by setting $M_0$ as the set representing the left-hand sides of $\mathcal{R}$, and the set $N_0$ as the set of ground instances of $f$ of the form $f(z_1, ..., z_n)$ where $f \in \mathcal{D}$. The algorithm then iteratively tries to unify elements of $N_0$ with the elements of $M_0$ with a substitution $\sigma$. In case such a substitution is found, the matched elements $m \in M$ and $n \in N$ are removed from $M_i$ and $N_i$ and new sets $M_{i+1}$ and $N_{i+1}$ are constructed by:

$$M_{i+1} = M_{i-1} \setminus \{m\} \cup \{m\rho \mid \rho \in C(\sigma), m\rho \neq m\sigma\}$$
$$N_{i+1} = N_{i-1} \setminus \{n\} \cup \{n\rho \mid \rho \in C(\sigma), n\rho \neq n\sigma\}$$

The complement of a substitution $\sigma$ defined as the set $C(\sigma)$ of all substitutions $\rho$ different from $\sigma$, having the same domain and mapping elements to

complementary term. The complement of a term $t$ is defined as the finite set such that the ground terms of $t$ and the ground terms of the set of complement terms equal the set of constructor terms.

The algorithm continues until $M_{last}$ or $N_{last}$ are empty (i.e. the last pair of $M$ and $N$ sets), or no further unification is possible. If both $M_{last}$ and $N_{last}$ are empty, $f$ is convertible to the constructors ($f$ covers all input terms and/or the complement can be further reduced by the rules). If $M_{last}$ is empty but $N_{last}$ is not empty, $f$ is not defined on the terms in $N_{last}$.

The set $N_{last}$ is interesting, as it contains the patterns not matched by $f$. It can, therefore, be used in a functional programming setting, warning the user of incomplete patterns (and hinting on actual constructor patterns currently unmatched).

From these, quasi-reducibility is determined when either each input term is matched by the LHS of $f$ (both $M_{last}$ and $N_{last}$ are empty), or the elements of $N_{last}$ can further be reduced.

Pattern completeness is given in case both $M_{last}$ and $N_{last}$ are empty.

## 4.1 Examples

**Example 4.1.** Let us take an example execution of the algorithm on the same input as Example 3.1.

$$\begin{aligned} M_0 &= \{f(0), f(succ(x))\} \\ N_0 &= \{f(z)\} \\ \Rightarrow M_1 &= \{f(succ(x))\} \\ N_1 &= \{f(succ(z))\} \\ \Rightarrow M_2 &= \varnothing \\ N_2 &= \varnothing \end{aligned}$$

The sets $M_0$ and $N_0$ are the starting steps, we would like to check whether $f(z)$ is convertible with the left-hand sides of the system. In the first iteration the case of $f(0)$ and $f(z)$ are unified by the substitution $\sigma = \{z \mapsto 0\}$. Then we pick a substitution $\rho$ from the set of complement substitutions $C(\sigma)$, take $\rho = \{z \mapsto succ(z)\}$. In the last step, we take $\sigma = \{z \mapsto x\}$. Since both $M_2$ and $N_2$ are empty, the definition of $f$ is said to be *relatively complete*.

**Example 4.2.** The counterpart of Example 3.2:

$$\begin{aligned} M_0 &= \{f(succ(x))\} \\ N_0 &= \{f(z)\} \\ \Rightarrow M_1 &= \varnothing \\ N_1 &= \{f(0)\} \end{aligned}$$

In the first step we try to unify $f(succ(x))$ with $f(z)$. This we can do via $\sigma = \{z \mapsto succ(x)\}$. Then we take $\rho$ from $C(\sigma)$ as $\rho = \{z \mapsto 0\}$, after which we try to unify $f(succ(x))$ and $f(0)$, which fails. In $N_1$ we find the term that is not covered by the definition of $f$ (it is not further reducable), therefore $f$ is not relatively complete.

**Example 4.3.** Finally, the quasi-reducible system $\mathcal{R}_{\mathbb{Z}}$ from Examples 2.2, 3.5:

$$\begin{aligned}
& M_0 = \{even(0), even(s(0)), even(s(s(x))), even(p(0)), even(p(p(x)))\} \\
& N_0 = \{even(z)\} \\
\Rightarrow\ & M_1 = \{even(s(0)), even(s(s(x))), even(p(0)), even(p(p(x)))\} \\
& N_1 = \{even(s(z)), even(p(z))\} \\
\Rightarrow\ & M_2 = \{even(s(s(x))), even(p(p(x)))\} \\
& N_2 = \{even(s(s(z))), even(s(p(z))), even(p(s(z))), even(p(p(z)))\} \\
\Rightarrow\ & M_3 = \varnothing \\
& N_3 = \{even(s(p(z))), even(p(s(z)))\}
\end{aligned}$$

Note the fact that $M_0$ does not contain left-hand sides $s(p(x))$ and $p(s(x))$ since the top-level function symbol is not even.

In the first step we unify via $\sigma = \{z \mapsto 0\}$, then apply $C(\sigma)$, i.e.: $\{z \mapsto s(z)\}$ and $\{z \mapsto p(z)\}$. We apply the same substitutions again to arrive at $M_2$. There, terms $even(s(s(x)))$ and $even(p(p(x)))$ are trivially matched by $even(s(s(z)))$ and $even(p(p(z)))$, therefore these pairs are removed. Finally, $M_3$ is empty so the algorithm stops. $N_3$ contains the patterns that are not defined for *even*, however, both these terms are further $\mathcal{R}_{\mathbb{Z}}$-reducible by the rules. Therefore, the definition of *even* is relatively complete. We can notice that the algorithm indirectly also proven pattern incompleteness, as $N_3$ contains patterns where the function *even* needs to be defined.

# 5 Tree automata-based algorithms

Pattern completeness of left-linear systems can also be verified using tree automata based solution, e.g. with the framework developed by Middeldorp et al. in [5] or by Bouhoula and Jacquemard in [6]. The experiments done by Thiemann and Yamada in [1] construct tree automata $\mathcal{A}$ and $\mathcal{B}$ for their test cases and verify the language inclusion problem $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ via the framework.

A tree automaton $\mathcal{A}$ over an alphabet $\mathcal{F}$ is defined as the 4-tuple $(Q, \mathcal{F}, Q_f, \Delta)$, where $Q$ is the set of states, $Q_f \subseteq Q$ are the final states, $\Delta$ are the transition rules between states. Transition rules are defined as the set of rules of the form $f(q_1(x_1), ..., q_n(x_n)) \to q(f(x_1, ..., x_n))$, where $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, ..., q_n \in Q$, $x_1, ..., x_n \in \mathcal{X}$. A term is accepted by $\mathcal{A}$ if $t \xrightarrow[\mathcal{A}]{*} q(t), q \in Q_f$. Bottom-up tree automata start their computation at the leaves of the tree and move upwards, in contrast with top-down tree automata which start at the root. The language

$\mathcal{L}(\mathcal{A})$ of tree automaton $\mathcal{A}$ is defined as the set of ground terms accepted by $\mathcal{A}$ [7].

To translate pattern problems to tree automata domain, the following construction can be used, as demonstrated in the paper of Thiemann and Yamada[1]. Firstly, for the term rewrite system two tree automata $\mathcal{A}$ and $\mathcal{B}$ are constructed. Tree automata $\mathcal{A}$ accepts each valid input of the term rewrite system, whereas tree automata $\mathcal{B}$ accepts each left hand side of the system. The pattern problem then reduces to the language inclusion problem $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Namely, that for each term recognised by tree automaton $\mathcal{A}$, there exist a matching term recognised by tree automaton $\mathcal{B}$. Conversely, if there is an input term recognised by tree automaton $\mathcal{A}$, but not recognised by tree automaton $\mathcal{B}$, then there is an incomplete pattern.

The framework by Middeldorp et al. [5] constructs bottom-up tree automata to verify properties thereof, whereas the algorithm by Bouhoula et al. [6] construct many-rooted top-down tree automata.

# 6 Further notable work

In [4], Thiel introduces calculus of components, on which the paper by Lazrek et al. is based. The complement of a term $t$ in $\mathcal{T}(\mathcal{C}, \mathcal{X})$ is defined as the finite set $T \subseteq \mathcal{T}(\mathcal{C}, \mathcal{X})$ such that $G(t) \cup G(T) = \mathcal{T}(\mathcal{C})$, i.e. the union ground terms of $t$ and $T$ equal the constructor ground terms. Their algorithm details a way to decide sufficient completeness, similar to the complement algorithm of [2].

Decidability of quasi-reducibility was shown by Kapur et al. in [3]. Their algorithm, however, is impractical in practice, as it has exponential best-case complexity. The *complement algorithm* by Lazrek et al. is a refinement of this paper.

In [8], Aoto and Toyama introduce *strong quasi-reducibility*, in their paper Ground Confluence Prover based on Rewriting Induction. Strong quasi-reducibility extends quasi-reducibility to term rewriting systems with non-free constructors, i.e. constructors that can be further reduced in the system.

Cynthia Kop presented and algorithm to decide quasi-reducibility in logically constrained term rewrite systems in [9]. These LCTRSs are of the nature e.g. "rule $x \to y$ is applicable only if $x > 5$".

Bouhoula and Jacquemard constructed a tree-automata based framework to decide sufficient completeness of logically constrained term rewrite systems in [6].

The Glasgow Haskell Compiler [10] performs pattern completeness checks by enabling `-Wincomplete-patterns`. It applies, however, only to linear patterns, as non-linear patterns like `f a a = ...` are not allowed by the language, they need to be simulated by guards like `f x y | x == y = ...`.

# 7 Comparison

Both algorithms presented by Thiemann and Yamada in [1] and by Lazrek et al. in [2] are able to decide whether a given term rewrite system is pattern complete. The focus of the complement algorithm is to conclude relative completeness, however, one can make use of $N_{last}$ to conclude whether the program is also pattern complete. Namely, when $N_{last}$ is not empty, the set contains the patterns where the program still needs to be defined. One notable difference between the two algorithms is that the refined version of Thiemann and Yamada's proven to work with non-linear patterns, whereas, the algorithm by Lazrek et al. might not. The paper by Lazrek et al. mentions certain examples of non-linearity where the algorithm successfully completes, but also in cases where it would get stuck.

Another aspect that makes the complement algorithm interesting is its ability of counterexample generation. By default the contents of $N_{last}$, after checking for irreducibility, contains the patterns where the function $f$ still needs to be defined[2]. The algorithm by Thiemann and Yamada, by default, does not have this ability – though it is mentioned as an easy improvement in the paper[1].

Tree automata have proven useful for deciding pattern completeness and related notions, but current algorithms e.g. in [5] are restricted to left-linear systems.

**Further questions** One question that might arise after reviewing the above papers, is that it remains to be seen how these algorithms would perform on a more exhaustive performance testing against each other. Namely, the examples created by Thiemann and Yamada clearly give the upper hand to their algorithm[1], however, their method of example generation seems a bit contrived. One might wonder how the algorithms would fare on more "typical" inputs such as small functional programs from existing projects. Moreover, the algorithms that Thiemann and Yamada's are checked against are not implemented or in any case tuned by the authors, but are being used from 3rd party tools such as the ground confluence prover of Aoto and Toyama [8], or the tree automata framework developed by Middeldorp et al.[5]. This fact might explain the constant factor performance difference between the runtimes.

Finally, as suggested by Thiemann and Yamada, it remains to be seen whether their algorithm can be adjusted to decide quasi-reducibility, or whether a similar syntax-directed algorithm can be constructed.

# 8 Conclusion

The paper set out to explore algorithms for deciding pattern completeness in term rewrite systems. Pattern completeness is the notion that given a term rewrite system with left hand sides $L$ and basic ground term $f(t)$, the term is matched by some left hand side $\ell \in L$. The notion of quasi-reducibility was

also introduced that relaxes the pattern completeness definition, allowing for matching to happen under the root.

The main focus of the literature review is to discuss the algorithm by Thiemann and Yamada[1] and compare and contrast it with the *complement algorithm* of Lazrek et al. [2]. Moreover, frameworks using tree automata are proven useful for deciding pattern completeness and related definitions. Therefore, a short introduction of this construction is also discussed. Finally, a short survey of related literate is also included at the end of Section **??**.

Further research, as discussed in Section 7, could explore a more detailed and exhaustive performance comparison of the discussed algorithms. Moreover, as per [1], it remains open to construct a similar syntax-based algorithm to decide quasi-reducibility.

# References


[1] René Thiemann and Akihisa Yamada. "A Verified Algorithm for Deciding Pattern Completeness". In: *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*. Ed. by Jakob Rehof. Vol. 299. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 27:1–27:17. ISBN: 978-3-95977-323-2. DOI: 10.4230/LIPIcs.FSCD.2024.27. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2024.27.

[2] Azeddine Lazrek, Pierre Lescanne, and Jean-Jacques Thiel. "Tools for proving inductive equalities, relative completeness, and $\omega$-completeness". *Information and Computation* 84.1 (1990), pp. 47–70. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(90)90033-E. URL: https://www.sciencedirect.com/science/article/pii/089054019090033E.

[3] Deepak Kapur, Paliath Narendran, and Hantao Zhang. "On sufficient-completeness and related properties of term rewriting systems". *Acta Inf.* 24.4 (Aug. 1987), pp. 395–415. ISSN: 0001-5903. DOI: 10.1007/BF00292110. URL: https://doi.org/10.1007/BF00292110.

[4] Jean Jacques Thiel. "Stop losing sleep over incomplete data type specifications". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: Association for Computing Machinery, 1984, pp. 76–82. ISBN: 0897911253. DOI: 10.1145/800017.800518. URL: https://doi.org/10.1145/800017.800518.

[5] Aart Middeldorp, Alexander Lochmann, and Fabian Mitterwallner. "First-Order Theory of Rewriting for Linear Variable-Separated Rewrite Systems: Automation, Formalization, Certification". 67.2 (Apr. 2023). ISSN: 0168-7433. DOI: 10.1007/s10817-023-09661-7. URL: https://doi.org/10.1007/s10817-023-09661-7.

[6] Adel Bouhoula and Florent Jacquemard. "Sufficient completeness verification for conditional and constrained TRS". *Journal of Applied Logic* 10.1 (2012). Special issue on Automated Specification and Verification of Web Systems, pp. 127–143. ISSN: 1570-8683. DOI: https://doi.org/10.1016/j.jal.2011.09.001. URL: https://www.sciencedirect.com/science/article/pii/S1570868311000413.

[7] Hubert Comon et al. *Tree Automata Techniques and Applications*. 2008, p. 262. URL: https://inria.hal.science/hal-03367725.

[8] Takahito Aoto and Yoshihito Toyama. "Ground Confluence Prover based on Rewriting Induction". In: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 33:1–33:12. ISBN: 978-3-95977-010-1. DOI: 10.4230/LIPIcs.FSCD.2016.33. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2016.33.

[9] Cynthia Kop. *Quasi-reductivity of Logically Constrained Term Rewriting Systems*. 2017. arXiv: 1702.02397 [cs.LO]. URL: https://arxiv.org/abs/1702.02397.

[10] GHC Team. *The Glasgow Haskell Compiler*. https://www.haskell.org/ghc/. Accessed: 2024-10-05. 2024.