Beyond Pattern Completeness: Deciding Quasi-Reducibility in Left-Linear TRSs

Vrije Universiteit Amsterdam

Master's Thesis

Daniel Koves

Supervisors
Femke van Raamsdonk
Jörg Endrullis

August 2025

Abstract

Pattern completeness, as a property of functional programs, guarantees that a function is exhaustively defined for all possible inputs of its type. When modelling functional programs via term rewrite systems, pattern completeness ensures that any *basic* term – a ground term that has a defined symbol at the root position – can be rewritten at the top.

We adapt the decision procedure of Thiemann and Yamada [1] for pattern completeness to address quasi-reducibility, a related notion to pattern completeness that ensures computation does not get stuck by permitting matches to happen anywhere in the term.

The adaptation has three variations: first one considers strong quasi-reducibility as per [2] as a stepping stone to arrive at the second variation addressing general quasi-reducibility. Finally, the last version adapts the algorithm for sorted applicative term rewriting. The correctness of the algorithms are demonstrated by a termination argument and a soundness argument.

Contents

Chapter 1

Introduction

Pattern completeness in the world of functional programming is a crucial notion ensuring that given any input a function can compute a result. This is done by ensuring that the function definition cover all cases of the input, as in, if the function takes e.g. a list of numbers, then the definition must cover all constructors of lists: the empty list and the non-empty list (or *cons*). Algorithms to decide pattern completeness (and similar notions) have existed for long time [3, 4, 5], indeed all general functional programming languages implement such an algorithm in their compilers, like the Glasgow Haskell Compiler for Haskell [6]. Most of these algorithms are implemented by translating the pattern completeness problem to a series of *matching problems*, where an input term is matched to each of the cases in the definition of the given function. If the input term can be matched to all the cases, then we can conclude that the given function is pattern complete. Example of an incomplete program can be seen on Figure 1.1, notice the missing case for `Nothing`:

```haskell
first :: [a] -> Maybe a
first (x:_) = Just x

main :: IO ()
main = print $ first []
```

Figure 1.1: Haskell with an incomplete pattern: this would blow up!

And the version that is complete can be seen on Figure 1.2:

```
first :: [a] -> Maybe a
first (x:_) = Just x
first []    = None

main :: IO ()
main = print $ first []
```

Figure 1.2: Haskell with a complete pattern: this is now fine

While pattern completeness is a useful, and quite strong requirement to have, in some other cases, one can make do with less strict notions. One such notion is quasi-reducibility, which captures the requirement that computation does not get stuck. While for pattern completeness, we needed input terms to be matched to the cases of the function definition, for quasi-reducibility, it is enough that a subterm of the input term can be matched to the cases of the definition. If we try to translate this to a real-world program, it would mean something like: while we can have incomplete function definitions, but as long as we can match the input to any left-hand side (the cases in the function definitions), we ensure that we always can compute a value. This can be modelled in functional languages by using free monads or free algebras, which lets us define custom evaluation strategies that can be useful in domain specific languages and tree-like settings [7].

One such pattern completeness algorithm, which works by translating the input problem into a series of matching problems is that of Thiemann and Yamada [1]. The paper models computation via term rewriting, which is a sufficiently interesting and fitting framework for such algorithms. Term rewriting is expressible enough to model such computations with the added benefit of being quite intuitive. We can translate the earlier Haskell example using the following term rewrite system in Example 1. The background for term rewriting is further discussed in Chapter 2.

Example 1. We define a term rewrite system $\mathcal{R}$ with a defined symbol $\mathcal{D} = \{\text{first}\}$ and constructors $\mathcal{C} = \{\texttt{[]}, \texttt{:}\}$. The rewrite rules of the system are as follows:

$$R = \{\text{first } (x : xs) \to x,\ \text{first } \texttt{[]} \to \texttt{[]}\}$$

Furthermore, for pattern completeness specifically, modelling the algorithms via term rewriting is also natural, since the algorithms we consider are syntactic in nature and programs written in functional languages can easily be adapted to term rewrite systems. As the title suggests, we will consider *left-linear* term rewrite systems, where a variable in the left-hand side of the rules cannot appear more than once. This is a natural restriction to make when we consider functional programming languages, as most languages only allow left-linear function definitions, moreover, this makes the algorithm easier to reason about.

Following the suggestion in the aforementioned paper, we develop a syntax-oriented algorithm to decide quasi-reducibility. Using the approach from the paper, we detail a modification of their left-linear algorithm for pattern completeness and adapt it for quasi-reducibility. Firstly, a variant for strong-quasi reducibility is detailed. Strong quasi-reducibility permits matching under the root, but not in any subterm. This modification lets us move past the initial clash of root symbols when the TRS has quasi-reducible rules. This detail is explained more in detail in Section 3.2. Further, the algorithm is adapted such that matches in any subterm is permitted, arriving at the general quasi-reducible solution. Lastly, we move past the standard first-order rewriting setting and introduce simple types and applicative syntax into the system, while not allowing for lambdas, therefore, keeping the matching algorithm decidable.

The thesis is organised as follows: Chapter 1 introduces the problem at hand and section 1.1 includes a short survey of literature. Chapter 2 discusses the mathematical background needed to describe and reason about the algorithms, that is, in the setting of term rewriting. Chapter 3 discusses the different versions of the linear algorithm for strong quasi-reducibility in section 3.2, general quasi-reducibility in section 3.3 and quasi-reducibility with applicative syntax in section 3.4. Chapter 4 delves into the correctness of the approach, where section 4.1 includes the termination, and section 4.2 the soundness arguments. Lastly, Chapter 5 gives the concluding remarks.

The main theorems of the paper, Theorem 1 for termination of the decision procedure and Theorem 3 for the soundness can be found in Chapter 4.

1.1 In Literature

The study of pattern completeness and related notions like relative completeness in the world of term rewriting have a long history dating back to the 80s. Following is a short survey of the literature, covering the span of the last 40 years, with some excerpts taken from a literature study conducted by the author of this paper [8].

In [3], Thiel introduces calculus of components. 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 Lazrek et al[5].

Decidability of quasi-reducibility was shown by Kapur et al. in [4]. 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.

The *complement algorithm* presented by Lazrek, Lescanne, and Thiel in [5], presents 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 **??**. 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 complement of a substitution σ defined as the set $C(\sigma)$ of all substitutions ρ different from σ, 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.

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 [9] or by Bouhoula and Jacquemard in [10]. 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.

In [2], 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. Moreover, strong quasi-reducibility, as it will be detailed in section 3.2, allows matching to happen only directly under the root.

Cynthia Kop presented and algorithm to decide quasi-reducibility in logically constrained term rewrite systems in [11]. 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 [10].

The Glasgow Haskell Compiler [6] 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 = ...`.

Chapter 2

Preliminaries

This chapter covers the mathematical background for Term Rewriting and matching problems. Since the algorithms we consider model computation via term rewriting, these notions are crucial to understand and will be used through the paper.

2.1 Term Rewriting

For untyped term rewriting, we assume a countably infinite set $\mathcal{X}$ of *variables* written as x, y, z. A *signature* Σ is a finite set consisting of function symbols written as f, g, h. Each function symbol comes with a natural number, called its *arity*, representing the number of arguments the function symbol takes. We write f if f has arity 0, and $f(x)$ if f has arity 1.

Given a signature Σ, we say that the set of terms over Σ denoted as $\mathcal{T}(\Sigma, \mathcal{X})$ is the smallest set such that:

1. If x in $\mathcal{X}$, then $x \in \mathcal{T}(\Sigma, \mathcal{X})$
2. If f in Σ with arity n $t_1, ..., t_n \in \mathcal{T}(\Sigma, \mathcal{X})$, then $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X})$

By $\mathcal{V}ar(t)$ we denote the set of variables in t. *Ground* terms are terms without variables. The set of ground terms over a signature Σ is denoted as $T(\Sigma)$. 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 $1 \leq i \leq n$.

Given a term t in $\mathcal{T}(\Sigma, \mathcal{X})$, the set of *positions* of t, denoted as $\mathcal{P}os(t)$, is defined recursively on the definition of a term:

1. If $t = x$ then $\mathcal{P}os(t) = \{\epsilon\}$
2. If $t = f(t_1, \ldots, t_n)$ then $\mathcal{P}os(t) = \{\epsilon\} \ \cup \bigcup_{i=1}^{i=n} \{ip \mid p \in \mathcal{P}os(t_i)\}$

If $p \in \mathcal{P}os(t)$, then by $t|_p$ we note the subterm of t at position p, and by $s[t]_p$, where we replace the term in s at position p by t.

A *substitution* σ is a mapping $\mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$. A substitution extended to a mapping from terms to terms is the result of applying substitution σ to a term t, denoted as $t\sigma$ (alternative notations include $\sigma(t)$ or t^σ). We say a term t matches term s of there exists a substitution σ such that $t\sigma = s$. We say terms t and s can be unified if there are substitutions σ and γ such that $t\sigma = s\gamma$. More about matching can be found in Section 2.2.

Given a signature Σ, a *rewrite rule* over Σ is a pair of terms (ℓ, r), denoted as $\ell \to r$ such that $\ell \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell)$. A *term rewrite system* $\mathcal{R}$ is the pair (Σ, R) where R is a set of rewrite rules between $\mathcal{T}(\Sigma, \mathcal{X})$.

A *rewrite relation* $\to_{\mathcal{R}}$ is a binary relation induced by the rewrite rules R. We say that a term s rewrites in one step to term t, denoted as $s \to_{\mathcal{R}} t$, given that there is a rule $\ell \to r \in R$, a position p in s and a substitution σ, we have that $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$. A *rewrite sequence* is a chain of rewrite steps $s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \dots$. Rewrite sequences can be finite or inifite. We write $\to_{\mathcal{R}}^*$ as the reflexive-transitive closure of $\to_{\mathcal{R}}$.

In a TRS $\mathcal{R}$ over Σ, we take $\Sigma = \mathcal{C} \cup \mathcal{D}$ where $\mathcal{D}$ represents *defined symbols* and $\mathcal{C}$ *constructors*. We consider inputs to functions as constructor ground terms. We say a term rewrite system is *left-linear*, if each of the variables on the left-hand side of the rewrite rules only appear once, i.e. a LHS $f(a, b)$ is allowed, whereas $f(a, a)$ is not.

For sorted term rewriting, we fix a set of sorts $\mathcal{S}$. Then a sorted set of variables $\mathcal{V}$ is a set in which each element $v \in \mathcal{V}$ 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 $\mathcal{T}(\mathcal{F}, \mathcal{V})$ similarly to above.

Example 2. Consider the following term rewrite system to add two natural numbers. We fix a set of sorts $\mathcal{S} = \{\mathbb{N}\}$ representing natural numbers.

We have a defined symbol $\mathcal{D} = \{add : \mathbb{N} \times \mathbb{N} \to \mathbb{N}\}$ and the following constructors for natural numbers in Peano arithmetic:

$$\mathcal{C} = \{0 : \mathbb{N},\ s : \mathbb{N} \to \mathbb{N}\}$$

Then our TRS can be defined using the following set of rules:

$$R = \left\{ \begin{matrix} add(0, y) \to y \\ add(s(x), y) \to s(add(x, y)) \end{matrix} \right\}$$

2.2 Matching Problems

The *matching problem* asks, given two terms s and t, whether there exists a substitution σ such that $s\sigma = t$. Different notations exist to represent matching problems, like $s \mapsto^? t$ or $s \lesssim^? t$. The direction is from s to t, namely, we try to match s to t by σ. A matching problem mp is represented as a set $\{(t_1 \mapsto^? \ell_1), ..., (t_n \mapsto^? \ell_n)\} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$, assuming $\mathcal{V}$ and $\mathcal{X}$ do not overlap. A pattern problem pp is a finite set of matching problems.

Definition 1. A program with LHSs L is pattern complete if every basic ground term is matched by some $\ell \in L$.

Definition 2. A program with LHSs L is quasi-reducible complete if every basic ground term contains a subterm that is matched by some $\ell \in L$.

Definition 3. A matching problem mp is a finite set $mp = \{(t_1 \mapsto^? \ell_1), \dots, (t_n \mapsto^? \ell_n)\}$ of pairs of terms. A pattern problem pp is a set of matching problems $pp = \{mp_1, \dots, mp_n\}$.

The matching algorithm of Thiemann and Yamada [1] is defined on a set of pattern problems. Namely, a if the set of pattern problems $P = \{\{\{(f(x_1, \dots, x_n) \mapsto^? \ell)\} \mid \ell \in L\} \mid f \in \mathcal{D}\}$ is complete, the program with defined symbol f and LHSs L is pattern complete.

Definition 4. A set of pattern problems P is complete if for each constructor ground substitution σ and all pattern problems $pp \in P$ there exists a substitution γ and matching problem $mp \in pp$ such that for all $(t \mapsto^? \ell) \in mp$ we have that $t\sigma = \ell\gamma$.

These definitions form the basis of the pattern completeness algorithm by Thiemann and Yamada [1]. Following sections in Chapter 3 will relax and adapt these definitions to arrive at an algorithm capable of deciding (different forms of) quasi-reducibility.

Chapter 3

Linear Algorithm for Quasi-Reducibility

This section shall explore relevant notions needed to arrive at the left-linear algorithm to decide quasi-reducibility. Starting from pattern completeness – the most strict definition - which is detailed in the paper by Thiemann and Yamada [1]. Following, rule modifications needed for strong quasi-reducibility is detailed – as per the paper of Aoto and Toyama [2] – where matching is permitted directly under the root. Finally, the modification of the algorithm is given, such that general quasi-reducibility can be decided, allowing for matches to happen anywhere under the root.

In the second half of the section, a version of the algorithm is detailed which uses Term Rewrite Systems with simple types and applicative syntax but no lambdas, allowing for a more generic TRSs while keeping the algorithm decidable and syntax-oriented, as it is still in the domain of first-order rewriting.

3.1 Pattern Completeness

The main rules of the linear algorithm for pattern completeness as per [1] are as follows:

Definition 5. Matching problems (denoted as mp) are reduced using the following rules:

$$\begin{array}{rl}
\textbf{decompose} & \{(f(t_1,\dots,t_n) \mapsto^? f(\ell_1,\dots,\ell_n))\} \uplus mp \to \{(t_1 \mapsto^? \ell_1),\dots,(t_n \mapsto^? \ell_n)\} \cup mp \\
\textbf{match} & \{(t \mapsto^? x)\} \uplus mp \to \varnothing \ \text{ if } \ \forall (t' \mapsto^? \ell) \in mp.\ x \notin \mathrm{Var}(\ell) \\
\textbf{clash} & \{(f(\dots) \mapsto^? g(\dots))\} \uplus mp \to \bot_{mp} \text{ if } f \neq g
\end{array}$$

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

remove-mp	$\{\bot_{mp}\} \uplus pp \to pp$
success	$\{\varnothing\} \uplus pp \to \varnothing$

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

failure	$\{\varnothing\} \uplus P \to \bot$
instantiate	$\{pp\} \uplus P \to \mathrm{Inst}(pp, x) \cup P$ if $\{(x, f(\dots))\} \in pp$

For (instantiate), whenever we have a matching problem of the form $\{(x \mapsto^? f(\dots))\}$, the algorithm would detect a clash. However, since x represents some arbitrary constructor ground term, we "instantiate" x by replacing it with each constructor of the form $c(x_1, \dots, x_n)$ for all $c \in \mathcal{C}$.

The set $\mathrm{Inst}(pp, x)$ generates the pattern problem:

$$\mathrm{Inst}(pp, x) = \{\{(t\sigma_{x,c} \mapsto^? \ell) \mid (t \mapsto^? \ell) \in mp\} \mid mp \in pp\}$$

given that $\sigma_{x,c} = [x \mapsto c(x_1, \dots, x_n)]$ for each constructor $c \in \mathcal{C}$ and fresh variables $x_1, \dots, x_n$.

This simplified version of the algorithm, which will be further adapted to address (strong) quasi-reducibility, can be seen in Examples 3, 5 using tree visualization. Each pattern problem (leaves) need to be closed by $\varnothing$, or else, if there is a pattern problem that is closed by $\bot$, the system is not complete (or quasi-reducible in the adaptation).

3.2 Strong Quasi-Reducibility

Strong quasi-reducibility, as per [2], is defined where matching is permitted in the direct subterms of the basic term. Strong quasi-reducibility naturally implies quasi-reducibility, since if we have strong quasi-reducibility, we have a direct subterm t' of input term t for each LHS $\ell \in L$. Then for quasi-reducibility we can also take subterm t' as a direct subterm is still a subterm of the input term t.

When we have a (strong) quasi-reducible system, we have rules whose LHS is rooted at a constructor (see also Example 3 and the section about quasi-reducibility 3.3). We use the strong quasi-reducibility adaptation to get past the initial clash of function symbols at the root and move the matching to the direct subterms.

To adapt the linear pattern completeness algorithm to decide strong quasi-reducibility, the following modifications of the rules from Definition 5 are made:

Definition 6. Rules for strong quasi-reducibility.

1. Whenever we have a matching problem of the form $\{(d(t_1,\dots,t_n) \mapsto^? \ell)\}$ with d a defined symbol, and (decompose) is not applicable, we apply a new rule (unwrap) and move the matching to the direct subterms of d:

   (unwrap): $\{(d(t_1,\dots,t_n) \mapsto^? \ell)\} \uplus mp \to \bigcup_{i\leq n}\{(t_i \mapsto^? \ell)\} \cup mp$ given $d \in \mathcal{D}$

2. Otherwise the matching problem clashes, when both terms are constructors and are not the same.

   (clash): $\{(c_1 \mapsto^? c_2)\} \uplus mp \to \bot_{mp}$ given $c_1, c_2 \in \mathcal{C}$ and $c_1 \neq c_2$

The unwrap rules needs a bit of explanation. When the arity of $d \in \mathcal{D}$ is 1, this rule can be also thought of as "lifting" the additional rules ℓ to the domain of the defined symbol d – the same result if the original matching problem was $\{(d(t_1,\dots,t_n) \mapsto^? d(\ell))\}$, and (decompose) would have been applicable, ensuring that the computation doesn't get stuck immediately.

Moreover, when the arity of d is ≥ 1, it splits the existing matching problem mp into n separate matching problems. This ensures that the matching moves to the direct subterms – but also that *any* subterm of $d(t_1,\dots,t_n)$ can match against ℓ. To visualize this, consider Example 4 and Figure 3.2.

Example 3. Consider a simple (strongly) quasi-reducible system with one defined symbol $f(.)$ and constructors 0 and $s(.)$. We write a program as a term-rewrite system:

$$f(0) \to 0$$

$$s(x) \to x$$

This system is (strongly) quasi-reducible, since for each basic ground term of the form $f(t)$ with t a constructor ground term, we can apply one of the rules. If t is of the form $f(s(.))$ then the second rule is applicable, otherwise the first one. The system is not pattern complete, since $f(s(.))$ is not covered by the rules.

The tree reduction of Example 3 can be seen on Figure 3.1. Due to the unwrap adaptation, we can move past the initial clash of $f(.)$ and $s(.)$, and close both constructor ground substitutions $z \mapsto 0$ and $z \mapsto s(z)$.

Example 4. Consider an example where the arity of the defined symbol is ≥ 1. Take TRS with defined symbol *add* and constructors 0, $s(.)$ and $g(.)$. The LHSs of the system are:

$$add(0,0) \quad add(0,s(x)) \quad add(s(x),0) \quad add(s(x),s(y)) \quad g(x)$$

$$\begin{tikzcd}
 & \{\{(f(z) \mapsto^? f(0))\}, \{(f(z) \mapsto^? s(x))\}\} \arrow[d, "\text{decompose}"] & \\
 & \{\{(z \mapsto^? 0)\}, \{(f(z) \mapsto^? s(x))\}\} \arrow[d, "\text{unwrap}"] & \\
 & \{\{(z \mapsto^? 0)\}, \{(z \mapsto^? s(x))\}\} \arrow[dl, "\begin{array}{c}\text{instantiate}\\ \sigma = \{x \mapsto 0\}\end{array}"'] \arrow[dr, "\begin{array}{c}\text{instantiate}\\ \sigma = \{x \mapsto s(z)\}\end{array}"] & \\
\{\{(0 \mapsto^? 0)\}, \{(0 \mapsto^? s(x))\}\} \arrow[d, "\text{match/clash}"] & & \{\{(s(z) \mapsto^? 0)\}, \{(s(z) \mapsto^? s(x))\}\} \arrow[d, "\text{clash/match}"] \\
\{\varnothing, \bot_{mp}\} \arrow[d, "\text{success}"] & & \{\bot_{mp}, \varnothing\} \arrow[d, "\text{success}"] \\
\varnothing & & \varnothing
\end{tikzcd}$$

Figure 3.1: Tree reduction of Example 3

Part of the tree reduction of Example 4 can be seen on Figure 3.2. Had we not split the matching problem $\{(add(a,b), g(x))\}$ into two matching problems $\{\{(a, g(x))\}, \{(b, g(x))\}\}$, the constructor ground substitution $\sigma = \{a \mapsto 0,\ b \mapsto g(z)\}$ would have failed.

$$\begin{tikzcd}[column sep=tiny]
 & & & \begin{array}{c}\{\{(\mathit{add}(a,b) \mapsto^? \mathit{add}(0,0))\}, \{(\mathit{add}(a,b) \mapsto^? \mathit{add}(s(x),0))\}, \{(\mathit{add}(a,b) \mapsto^? \mathit{add}(0,s(x)))\}, \\ \{(\mathit{add}(a,b) \mapsto^? \mathit{add}(s(x),s(y)))\}, \{(\mathit{add}(a,b) \mapsto^? g(x))\}\}\end{array} \arrow[d, "\text{decompose/unwrap}"] & & \\
 & & & \begin{array}{c}\{\{\{(a \mapsto^? 0),(b \mapsto^? 0)\}, \{(a \mapsto^? 0),(b \mapsto^? s(x))\}, \{(a \mapsto^? s(x)),(b \mapsto^? 0)\}, \\ \{(a \mapsto^? s(x)),(b \mapsto^? s(y))\}, \{(a \mapsto^? g(x))\}, \{(b \mapsto^? g(x))\}\}\end{array} \arrow[dl, "\begin{array}{c}\text{instantiate}\\ \sigma = \{a \mapsto 0\}\end{array}"'] \arrow[dr] \arrow[drr] & & \\
 & & \begin{array}{c}\{\{\{(0 \mapsto^? 0),(b \mapsto^? 0)\}, \{(0 \mapsto^? 0),(b \mapsto^? s(x))\}, \{(0 \mapsto^? s(x)),(b \mapsto^? 0)\}, \\ \{(0 \mapsto^? s(x)),(b \mapsto^? s(y))\}, \{(0 \mapsto^? g(x))\}, \{(b \mapsto^? g(x))\}\}\end{array} \arrow[dll] \arrow[dl] \arrow[dr, "\begin{array}{c}\text{instantiate}\\ \sigma = \{b \mapsto g(z)\}\end{array}"] & & \dots & \dots \\
\dots & \dots & & \begin{array}{c}\{\{\{(0 \mapsto^? 0),(g(z) \mapsto^? 0)\}, \{(0 \mapsto^? 0),(g(z) \mapsto^? s(x))\}, \{(0 \mapsto^? s(x)),(g(z) \mapsto^? 0)\}, \\ \{(0 \mapsto^? s(x)),(g(z) \mapsto^? s(y))\}, \{(0 \mapsto^? g(x))\}, \{(g(z) \mapsto^? g(x))\}\}\end{array} & &
\end{tikzcd}$$

Figure 3.2: Tree reduction of Example 4

3.3 Quasi-Reducibility

To adapt the modified algorithm to handle (general) quasi-reducibility, we need to permit it to handle matches anywhere below the root. The modification from the previous section (i.e. the unwrap rule) is still needed, since the algorithm is syntax-oriented, and we do not want to get stuck when the outermost function symbols (one being a defined symbol of the basic ground term) clash.

Furthermore, we note that all rules, apart from instantiate, handle the matching of the outermost layer of the term. Whenever all rules have been exhaustively applied, we move to the subterms via the instantiate rule. By the instantiate rule, we get multiple pattern problems, one for each constructor, which allow for further matches and clashes to be applied.

In order to adapt the algorithm for quasi-reducibility, we would like to extract those LHSs of the rewrite rules of the term rewrite system, which are rooted at a constructor symbol. We then attach these patterns to the resulting pattern problem after instantiate, to allow for the newly instantiated constructors to match against them at every level of the term.

Therefore, we adapt the instantiate rule:

Definition 7. Instantiate rule for quasi-reducibility

$$\{pp\} \uplus P \Rrightarrow \mathrm{Inst}(pp, x) \cup \mathrm{Pat}(pp, x) \cup P$$

where Pat(pp, x), given thet set of LHSs L, is defined as the set:

$$\text{Pat}(pp, x) = \{\{(t\sigma_{x,c} \mapsto^? \ell)\} \mid \ell \in L, \text{root}(\ell) \notin \mathcal{D}\}$$

given that $\sigma_{x,c} = [x \mapsto c(x_1, \ldots, x_n)]$ for each constructor $c \in \mathcal{C}$ and fresh variables $x_1, \ldots, x_n$. The set Inst(pp, x) still generates the pattern problem:

$$\text{Inst}(pp, x) = \{\{(t\sigma_{x,c} \mapsto^? \ell) \mid (t \mapsto^? \ell) \in mp\} \mid mp \in pp\}$$

To illustrate why this works, consider the following lemma:

Lemma 1. Quasi-reducibility and pattern completeness coincide when all left-hand sides are basic terms, i.e. all left-hand sides are rooted at a defined symbol. Given a TRS $\mathcal{R}$ with signature $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ such that $\forall\ l \to r \in \mathcal{R}\ .\ \text{root}(l) \in \mathcal{D}$, we have that $\mathcal{R}$ is quasi-reducible $\iff$ $\mathcal{R}$ is pattern-complete.

Proof Pattern completeness implies quasi-reducibility follows from the definition, since t is a subterm of itself. Given that the system is quasi-reducible, then for all basic ground term $t = f(t_1, \ldots, t_n)$, there's a position p such that $t|p$ matches some left-hand side $\ell \in L$. Since all lhss ℓ have a root in $\mathcal{D}$, the only subterm $t|_p$ that can match a rule ℓ must be the term itself.

As we work with basic terms, the only rules that make the system quasi-reducible and not pattern complete are the ones whose LHS are rooted at a constructor. Otherwise when all LHSs of the rules are basic terms themselves, pattern completeness coincides with quasi-reducibility. Therefore, we make those constructor-headed LHSs available every step whenever a pattern problem is generated - at the beginning, or by instantiate. In other words, the constructor-headed LHSs are available in the reduction up to point where the first instantiate rule is applied. After that, we attach them manually whenever a new pattern problem is generated by instantiate, to allow for matches anywhere below the root.

Example 5. Given the following quasi-reducible system with one defined symbol $f(.)$ and constructors 0, $s(.)$ and $g(.)$. The LHSs are:

$$f(0) \quad f(s(0)) \quad f(s(s(x))) \quad g(x)$$

This system is quasi-reducible, since the first three rules cover the constructors 0 and $s(.)$, while the last rule is applicable whenever we have $g(.)$. The system is not pattern complete, since $f(g(.))$ is not covered by the rules. The system is not strongly quasi-reducible, since $f(s(g(s(.))))$ cannot be reduced by allowing matches only in the direct subterms of the term.

The reduction of the pattern problem generated in Example 5 can be seen on Figure 3.3. The patterns of the system are Pat$(pp, x) = \{g(x)\}$. After the first

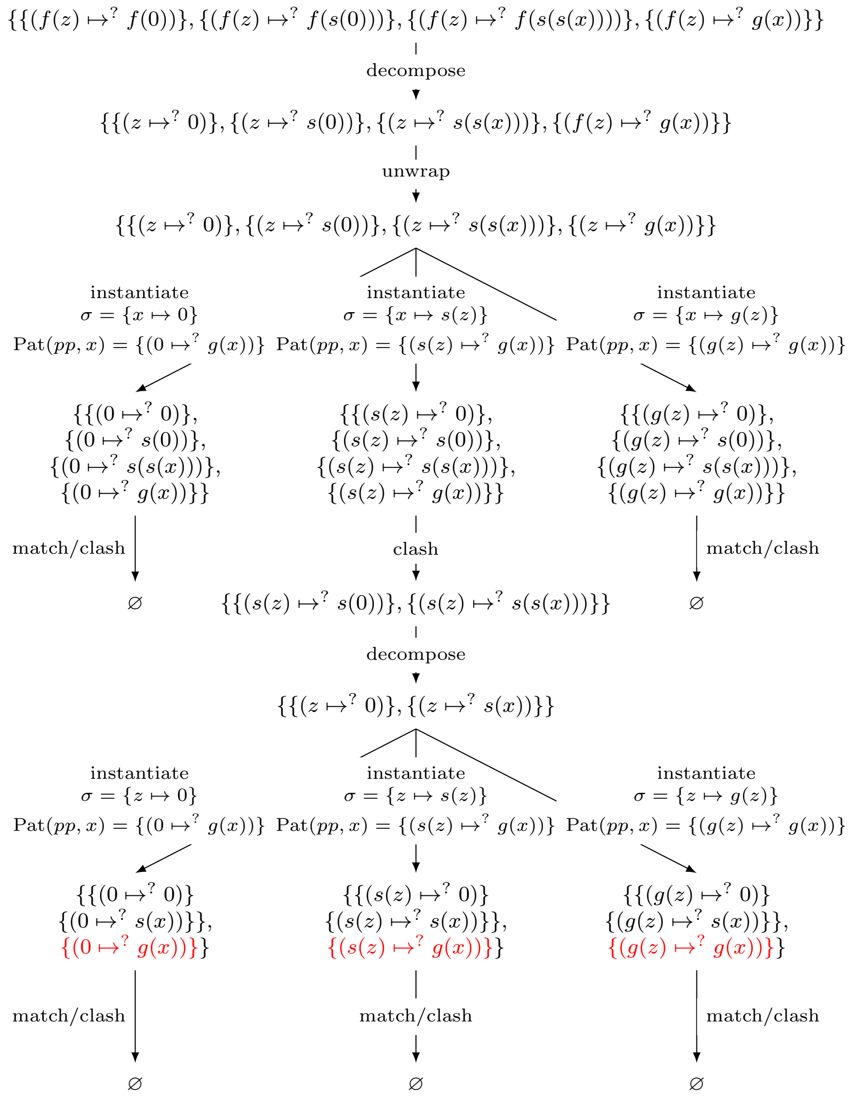
$\{\{(f(z) \mapsto^? f(0))\}, \{(f(z) \mapsto^? f(s(0)))\}, \{(f(z) \mapsto^? f(s(s(x))))\}, \{(f(z) \mapsto^? g(x))\}\}$
decompose
$\{\{(z \mapsto^? 0)\}, \{(z \mapsto^? s(0))\}, \{(z \mapsto^? s(s(x)))\}, \{(f(z) \mapsto^? g(x))\}\}$
unwrap
$\{\{(z \mapsto^? 0)\}, \{(z \mapsto^? s(0))\}, \{(z \mapsto^? s(s(x)))\}, \{(z \mapsto^? g(x))\}\}$
instantiate $\sigma = \{x \mapsto 0\}$ $\mathrm{Pat}(pp, x) = \{(0 \mapsto^? g(x))\}$
instantiate $\sigma = \{x \mapsto s(z)\}$ $\mathrm{Pat}(pp, x) = \{(s(z) \mapsto^? g(x))\}$
instantiate $\sigma = \{x \mapsto g(z)\}$ $\mathrm{Pat}(pp, x) = \{(g(z) \mapsto^? g(x))\}$
$\{\{(0 \mapsto^? 0)\}, \{(0 \mapsto^? s(0))\}, \{(0 \mapsto^? s(s(x)))\}, \{(0 \mapsto^? g(x))\}\}$
$\{\{(s(z) \mapsto^? 0)\}, \{(s(z) \mapsto^? s(0))\}, \{(s(z) \mapsto^? s(s(x)))\}, \{(s(z) \mapsto^? g(x))\}\}$
$\{\{(g(z) \mapsto^? 0)\}, \{(g(z) \mapsto^? s(0))\}, \{(g(z) \mapsto^? s(s(x)))\}, \{(g(z) \mapsto^? g(x))\}\}$
match/clash
clash
match/clash
$\varnothing$
$\{\{(s(z) \mapsto^? s(0))\}, \{(s(z) \mapsto^? s(s(x)))\}\}$
$\varnothing$
decompose
$\{\{(z \mapsto^? 0)\}, \{(z \mapsto^? s(x))\}\}$
instantiate $\sigma = \{z \mapsto 0\}$ $\mathrm{Pat}(pp, x) = \{(0 \mapsto^? g(x))\}$
instantiate $\sigma = \{z \mapsto s(z)\}$ $\mathrm{Pat}(pp, x) = \{(s(z) \mapsto^? g(x))\}$
instantiate $\sigma = \{z \mapsto g(z)\}$ $\mathrm{Pat}(pp, x) = \{(g(z) \mapsto^? g(x))\}$
$\{\{(0 \mapsto^? 0)\} \{(0 \mapsto^? s(x))\}\}, \{(0 \mapsto^? g(x))\}\}$
$\{\{(s(z) \mapsto^? 0)\} \{(s(z) \mapsto^? s(x))\}\}, \{(s(z) \mapsto^? g(x))\}\}$
$\{\{(g(z) \mapsto^? 0)\} \{(g(z) \mapsto^? s(x))\}\}, \{(g(z) \mapsto^? g(x))\}\}$
match/clash
match/clash
match/clash
$\varnothing$
$\varnothing$
$\varnothing$

Figure 3.3: Tree reduction of Example 5

instantiate, this pattern is still available in all resulting pattern problem. The algorithm clears the terms of the form $f(0)$ and $f(g(.))$. However, after the second instantiate, we already moved past the initial pattern problem, therefore the LHSs $g(x)$ is not available anymore, and we would have failed to match patterns of the form $f(s(g(\dots)))$. By attaching $\mathrm{Pat}(pp, x)$ to the pattern problem generated at the instantiate step, we clear the terms of the form $f(s(.))$. The crucial parts are highlighted red, the attached pattern of $g(x)$ makes the system quasi-reducible.

3.4 Applicative Term Rewriting

From this section onwards we consider sorted term-rewriting. Moreover, we extend the language of a term, by allowing applicative syntax (which we take as left-associative):

```
t ::= x | f | f t
```

We adapt the rules of the quasi-reducibility algorithm to handle applicative:

Definition 8. Applicative decompose (decompose-app):

$$\{(f\ t_1 \dots t_n \mapsto^? f\ \ell_1 \dots \ell_n)\} \uplus mp \to \{(f\ t_1 \dots\ t_{n-1} \mapsto^? f\ \ell_1 \dots\ \ell_{n-1}), (t_n \mapsto^? \ell_n)\} \cup mp$$

Definition 9. Applicative unwrap (unwrap-app):

$$\{(d\ t_1\ \dots\ t_n \mapsto^? \ell)\} \uplus mp \to \begin{cases} \text{Unwrap}(d\ \cdots \mapsto^? \ell) \cup mp & \text{if the set Unwrap is non-empty} \\ \bot_{mp} & \text{otherwise} \end{cases}$$

such that $\text{Unwrap}(d\ t_1\ \dots\ t_n \mapsto^? \ell) = \bigcup_{i=1}^{n} \{(t_i \mapsto^? \ell) \mid t_i : \iota \wedge \ell : \iota\}$

These rules mirror the non-applicative setting, we apply unwrap-app when decompose-app is not applicable, i.e. when there is a clash in d and $\text{root}(\ell)$. Moreover, since we're in a typed setting, we only allow unwrap-app to create well-typed matching problems, otherwise the matching problem rewrites to $\bot_{mp}$. Furthermore, we only allow well-typed matching problems to be generated by instantiate. Therefore, we make clear in the definition, that only those constructors and LHSs can be added to the sets *Inst* and *Pat*, that match the type of the variable to be instantiated:

Definition 10. Well-typed instantiation rule: given $x : \iota_0 \in \mathcal{V}$, where $\text{Pat}(pp, x)$, given thet set of LHSs L, is defined as the set:

$$\text{Pat}(pp, x) = \{\{(t\sigma_{x,c} \mapsto^? \ell)\} \mid \ell \in L,\ \text{root}(\ell) \notin \mathcal{D},\ \ell : \iota_0\}$$

given that $\sigma_{x,c} = [x \mapsto c(x_1, \dots, x_n)]$ for each constructor $c : \iota_1 \to \cdots \to \iota_n \to \iota_0 \in \mathcal{C}$ and fresh variables $x_1 : \iota_1, \dots, x_n : \iota_n \in \mathcal{V}$. The set $\text{Inst}(pp, x)$ still generates the pattern problem:

$$\text{Inst}(pp, x) = \{\{(t\sigma_{x,c} \mapsto^? \ell) \mid (t \mapsto^? \ell) \in mp\} \mid mp \in pp\}$$

Moreover, note that we also allow applicative constructors by letting $c : \iota_1 \to \cdots \to \iota_n \to \iota_0 \in \mathcal{C}$.

Consider the following example of pattern-complete TRS with applicative:

Example 6. Take sort of **List** $\mathbb{N}$ and constructors Nil : **List** $\mathbb{N}$ and *cons* : $\mathbb{N} \to$ **List** $\mathbb{N} \to$ **List** $\mathbb{N}$. Define the function *map* : $(\mathbb{N} \to \mathbb{N}) \to$ **List** $\mathbb{N} \to$ **List** $\mathbb{N}$:

$$\begin{aligned} map\ f\ \mathrm{Nil} &\to \mathrm{Nil} \\ map\ f\ (cons\ n\ l) &\to cons\ (f\ n)\ (map\ f\ l) \end{aligned}$$

We show that the map function is pattern complete (therefore also quasi-reducible), the reduction can be seen on Figure 3.4.

$$\begin{tikzcd}
& \{\{(map\ f'\ l' \mapsto^{?} map\ f\ Nil)\}, \{(map\ f'\ l' \mapsto^{?} map\ f\ (cons\ n\ l))\}\} \arrow[d, "\text{decompose-app}"] & \\
& \{\{(map\ f' \mapsto^{?} map\ f), (l' \mapsto^{?} Nil)\}, \{(map\ f' \mapsto^{?} map\ f), (l' \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{decompose-app}"] & \\
& \{\{(map \mapsto^{?} map), (f' \mapsto^{?} f), (l' \mapsto^{?} Nil)\}, \{(map \mapsto^{?} map), (f' \mapsto^{?} f), (l' \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{match}"] & \\
& \{\{(l' \mapsto^{?} Nil)\}, \{(l' \mapsto^{?} cons\ n\ l)\}\} \arrow[dl, "{\text{instantiate}\ \sigma = \{l' \mapsto Nil\}}"'] \arrow[dr, "{\text{instantiate}\ \sigma = \{l' \mapsto cons\ n'\ l'\}}"] & \\
\{\{(Nil \mapsto^{?} Nil)\}, \{(Nil \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{match/clash}"] & & \{\{(cons\ n'\ l' \mapsto^{?} Nil)\}, \{(cons\ n'\ l' \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{clash/decompose-app}"] \\
\varnothing & & \{\{(cons\ n' \mapsto^{?} cons\ n), (l' \mapsto^{?} l)\}\} \arrow[d, "\text{decompose-app}"] \\
& & \{\{(cons \mapsto^{?} cons), (n' \mapsto^{?} n), (l' \mapsto^{?} l)\}\} \arrow[d, "\text{match}"] \\
& & \varnothing
\end{tikzcd}$$

Figure 3.4: Tree reduction of Example 6

Moreover, consider a quasi-reducible version of this system:

Example 7. Quasi-reducible map with LHSs:

$$\{map\ f\ \mathrm{Nil},\ cons\ n\ l\}$$

We show that the system is quasi-reducible, the reduction can be seen on Figure 3.5.

$$\begin{tikzcd}
& \{\{(map\ f'\ l' \mapsto^{?} map\ f\ Nil)\},\{(map\ f'\ l' \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{decompose-app}"] & \\
& \{\{(map \mapsto^{?} map),(f' \mapsto^{?} f),(l' \mapsto^{?} Nil)\},\{(map\ f'\ l' \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{match}"] & \\
& \{(l' \mapsto^{?} Nil)\},\{(map\ f'\ l' \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{unwrap-app}"] & \\
& \{\{(map \mapsto^{?} map),(f' \mapsto^{?} f),(l' \mapsto^{?} Nil)\},\{(l' \mapsto^{?} cons\ n\ l)\}\} \arrow[dl, "\text{instantiate}\ \sigma = \{l' \mapsto Nil\}"'] \arrow[dr, "\text{instantiate}\ \sigma = \{l' \mapsto cons\ n'\ l'\}"] & \\
\{\{(Nil \mapsto^{?} Nil)\}\ \{(Nil \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{match}"'] & & \{\{(cons\ n'\ l' \mapsto^{?} Nil)\}\ \{(cons\ n'\ l' \mapsto^{?} cons\ n\ l)\}\} \arrow[d, "\text{match}"'] \\
\varnothing & & \varnothing
\end{tikzcd}$$

Figure 3.5: Tree reduction of Example 7

Chapter 4

Correctness

This section shows our two main results: the termination of the quasi-reducibility algorithm in Section 4.1 with Theorem 1, and the soundness of the algorithm in Section 4.2 with Theorem 3.

4.1 Termination

To prove termination, building on the argumentation in [1], we will construct a measure representing the "size" of a pattern problem by counting function symbols and show that each rule weakly decreases this measure while the instantiate rule strictly decreases it. Therefore, the instantiate rule cannot be applied infinitely often. The other rules match and clash are easily shown to be terminating, since they decrease the number of function symbols. Lastly, rules decompose and unwrap can also only be applied finitely often, as we assume terms to be finite.

Definition 11. Measure for size of pattern problems. Define $|(t \mapsto^? \ell)|$ as the size of a matching problem $(t \mapsto^? \ell) \in mp$, cases to be evaluated in order:

1. $|(x \mapsto^? \ell)|$ = number of functions symbols in ℓ given $x \in \mathcal{V}ar$
2. $|(f(t_1, \ldots, t_n) \mapsto^? f(\ell_1, \cdots \mapsto^? \ell_n))| = \sum_{i=1}^{n} |(t_i \mapsto^? \ell_i)|$
3. $|(d(t_1, \ldots, t_n) \mapsto^? \ell)| = \sum_{i=1}^{n} |(t_i \mapsto^? \ell)|$ given $d \in \mathcal{D}$
4. $|(t \mapsto^? \ell)| = 0$ otherwise

These four cases correspond to rules (instantiate) (1), (decompose) (2), (unwrap) (3) and (match) or (clash) (4).

Special matching problems like $\perp_{mp}$ and the empty problem $\varnothing$ have by definition measure 0.

For pattern problems pp, define $|pp| = \sum_{mp \in pp} \sum_{(t \mapsto^? \ell) \in mp} |(t \mapsto^? \ell)|$

For sets of pattern problems P, we write $|P|$ as the size of a set of pattern problems. We define relation $\succ$ on sets of pattern problems P via the multiset extension $>^{mul}$ of $>$ as:

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

By construction, all rules have the same measure before and after application, except the instantiate rule. The idea behind the measure for instantiate, is to replace the element in the multiset by one or more strictly smaller elements. Consider the following lemma:

Lemma 2. Instantiate rule strictly decreases with respect to $|(t \mapsto^? \ell)|$.

Proof To prove that the instantiate rule strictly decreases the measure, we work case-by-case. When (instantiate) is applicable, the pattern problem is of the form $P = \{\{\{(x \mapsto^? \ell)\}, \dots\}\}$ where $x \in \mathcal{V}ar$ and $\ell \notin \mathcal{V}ar$. After (instantiate) is applied, we get the following set of pattern problems of the form: $P' = \{\{\{(c(\dots) \mapsto^? \ell)\}\} \mid c \in \mathcal{C}\}$, i.e. one pattern problem per constructor, generated by the sets Inst and Pat. Assume that the cardinality of P is n, then the cardinality of P' is $n * |\mathcal{C}|$. Since the new pattern problems are of the form $\{\{(c(\dots) \mapsto^? \ell)\}\}$ for both pattern problems from Inst and Pat, only cases (2) or (4) are applicable in the measure definition. In case (2), when (decompose) is applicable, the resulting matching problem consists of a number of function symbols one less than before. Lastly, when case (4), i.e. (match) or (clash) are applicable, the resulting matching problem has measure 0 which is strictly less than the measure of $\{(x \mapsto^? \ell)\}$ if (instantiate) was applied. Therefore, the (instantiate) rule strictly decreases the measure by replacing an element of the multiset by one or more smaller elements.

Example 8. Measure for match rule

$$P = \{\{\{(0 \mapsto^? x)\}\}\} \Rrightarrow \{\{\varnothing\}\} = P'$$
$$|P| = \{|(0 \mapsto^? x)| = 0\} = |P'| \implies |P| \succeq |P'|$$

Measure for clash rule

$$P = \{\{\{(0 \mapsto^? s(x))\}\}\} \Rrightarrow \{\{\{\bot_{mp}\}\}\} = P'$$
$$|P| = \{|(0 \mapsto^? s(x))| = 0\} = |P'| \implies |P| \succeq |P'|$$

Measure for unwrap rule

$$P = \{\{\{(even(z) \mapsto^? s(p(x)))\}\}\} \Rrightarrow \{\{\{(z \mapsto^? s(p(x)))\}\}\} = P'$$
$$|P| = \{|(z \mapsto^? s(p(x)))| = 2\} = |P'| \implies |P| \succeq |P'|$$

Measure for unwrap rule with arity ≥ 1

$$P = \{\{\{(add(a,b) \mapsto^? g(x))\}\}\} \Rrightarrow \{\{\{(a \mapsto^? g(x))\}, \{(b \mapsto^? g(x))\}\}\} = P'$$
$$|P| = \{|(a \mapsto^? g(x))| + |(b \mapsto^? g(x))| = 2\} = |P'| \implies |P| \succeq |P'|$$

Measure for decompose rule

$$P = \{\{\{(f(a) \mapsto^? f(0))\}\}\} \Rrightarrow \{\{\{(a \mapsto^? 0)\}\}\} = P'$$
$$|P| = \{|(a \mapsto^? 0)| = 1\} = |P'| \implies |P| \succeq |P'|$$

Measure for instantiate rule

$$P = \{\{\{(a \mapsto^? 0)\}, \{(a \mapsto^? s(x))\}\}\}$$
$$\Rrightarrow \{\{\{(0 \mapsto^? 0)\}, \{(0 \mapsto^? s(x))\}\},\ \{\{(s(z) \mapsto^? 0)\}, \{(s(z) \mapsto^? s(x))\}\}\} = P'$$
$$|P| = \{2\},\ |P'| = \{0, 0\} \implies |P| \succ |P'|$$

We can now state the main theorem of this section:

Theorem 1. The decision procedure for quasi-reducibility in Section 3.3 is terminating.

Proof See Lemma 2 and paragraph above Definition 11.

4.2 Soundness

To prove soundness of the algorithm, we detail what it means for a pattern problem to be *quasi-reducible* and show that that property is preserved by the rewrite rules.

Definition 12. A set of pattern problems P is *quasi-reducible*, if for every constructor ground substitution σ and pattern problem $pp \in P$, there is some matching problem $mp \in pp$ and a substitution γ, such that for every pair $(t \mapsto^? \ell) \in mp$ either:

- $t\sigma = \ell\gamma$, or
- there exists a position p such that the subterm $t\sigma|_p = \ell\gamma$

Theorem 2. Let A be the modified algorithm for quasi-reducibility, and P the input pattern problem. Then we have that P is quasi-reducible iff $P \xrightarrow{A} P'$ is quasi-reducible.

Proof We show case by case that the quasi-reducibility of P is preserved by the rules.

Case - match

Given quasi-reducible pattern problem *pp*, fix constructor ground substitution σ. By quasi-reduciblity, we have a matching problem $mp \in pp$ and substitution γ such that for all $\{(t \mapsto^? \ell)\} \in mp$ we have that $t\sigma = \ell\gamma$ or some subterm matches. Before applying match we have some $(t \mapsto^? x) \in mp$ such that $t\sigma = x\gamma$. Therefore, we can take $\gamma = \{x \mapsto t\sigma\}$. Let pp' be the pattern problem after removing *mp* from *pp*. Then pp' is quasi-reducible iff. *pp* is quasi-reducible.

Case - clash

Given quasi-reducible pattern problem *pp*, fix constructor ground substitution σ. Take $mp \in pp$. Before applying clash, we have some $\{(f(.) \mapsto^? g(.))\} \in mp$ such that $f \neq g$. Therefore, the matching problem *mp* is incomplete. By quasi-reducibility of *pp*, we have some $mp' \neq mp \in pp$ that is quasi-reducible. Let pp' be the pattern problem after removing *mp* from *pp*. Then pp' is quasi-reducible iff. *pp* is quasi-reducible.

Case - decompose

Given quasi-reducible pattern problem *pp*, fix constructor ground substitution σ. Take $mp \in pp$. Before applying decompose we have some $\{(f(t_1, \ldots, t_n) \mapsto^? f(\ell_1, \ldots, \ell_n))\} \in mp$. By quasi-reducibility we have that $f(t_1, \ldots, t_n)\sigma = f(\ell_1, \ldots, \ell_n)\gamma$ or some subterm matches. This implies that for all $t_i\sigma$ we have corresponding $\ell_i\gamma$ such that $t_i\sigma = \ell_i\gamma$. After applying decompose we have $mp' = \{(t_1 \mapsto^? \ell_1), \ldots, (t_n \mapsto^? \ell_n)\}$. mp' is complete iff. *mp* is complete. Then pp' is quasi-reducible iff. *pp* is quasi-reducible.

Case - unwrap

Given matching problem $mp = \{(d(t_1, \ldots, t_n) \mapsto^? \ell)\}$ such that $\text{root}(\ell) \notin \mathcal{D}$, then the pattern problem $pp \ni mp$ is quasi-reducible iff. there exists a position p and substitution γ such that $d(t_1\sigma, \ldots, t_n\sigma)|_p = \ell\gamma$, since $\text{root}(d) \neq \text{root}(\ell)$.

After unwrap we get $\{(d(t_1, \ldots, t_n) \mapsto^? \ell)\} \rightarrow \{\{(t_1 \mapsto^? \ell)\}, \ldots, \{(t_n \mapsto^? \ell)\}\} = pp'$. If $d(t_1\sigma, \ldots, t_n\sigma)|_p = \ell\gamma$, then there exists a subterm

$t_i\sigma$ such that $t_i\sigma|_p = \ell\gamma$. Therefore, the resulting pattern problem pp' is quasi-reducible, iff. the original pattern problem pp is quasi-reducible.

Case - instantiate

Assume we have $pp = \{\{(x \mapsto^? \ell)\}\}$ before applying the instantiate rule, and fix constructor ground substitution σ. The rule rewrites pattern problem pp to $pp' = \mathrm{Inst}(pp, x) \ \cup \ \mathrm{Pat}(pp, x)$. The set Inst covers the outermost constructors, whereas the set Pat covers all constructor-headed LHSs. Assume that after applying the rule, there exists some $mp \in pp$ such that we have $(t\sigma \mapsto^? \ell) \in mp$.

Case I: Assume $\mathrm{Inst}(pp, x)$ generated a complete pattern problem pp' given constructor ground substitution σ such that there exists $mp' \in pp'$ and for all $(t \mapsto^? \ell) \in mp'$ we have that $t\sigma = \ell\gamma$. Then pp' is quasi-reducible iff. pp is quasi-reducible.

Case II: Assume $\mathrm{Pat}(pp, x)$ generated a complete pattern problem pp' given constructor ground substitution σ such that there exists $mp' \in pp'$ and for all $(t \mapsto^? \ell) \in mp'$ we have that $t\sigma = \ell\gamma$. Then pp' is quasi-reducible iff. pp is quasi-reducible.

Case III: Pattern problem generated pp' generated by $\mathrm{Inst}(pp, x) \cup \mathrm{Pat}(pp, x)$ is incomplete, therefore $t\sigma \neq \ell\gamma$. Then the original pattern problem cannot be quasi-reducible.

From Lemma 1 and the Definition 7 of Pat, it follows that there are no other cases, i.e. if a pattern problem is quasi-reducible, after an instantiate step, we get a pattern problem that is either complete by the set Inst, or complete by the set Pat, otherwise the pattern problem is not quasi-reducible. In other words, if the TRS is pattern complete, the set Pat is empty and the algorithm correctly verifies quasi-reducibility. Furthermore, when the TRS is quasi-reducible, the quasi-reducibility of the system is introduced by the rules in the set Pat.

We can now state the main theorem of this section:

Theorem 3. The decision procedure for quasi-reducibility in Section 3.3 is sound.

Proof See Theorem 2 and paragraph below the proof.

Chapter 5

Conclusion

The thesis has presented an extension of the pattern completeness algorithm by Thiemann and Yamada [1]. The original paper dealt with pattern completeness, in some sense, one of the strictest guarantee one can have regarding the correctness of a function. By ensuring pattern completeness we not only ensure that computation doesn't get stuck, but also that our function computes a value for all the possible input. A more relaxed notion of it is quasi-reducibility, in which we permit matches below the root.

The thesis presented three variations of the algorithm: one dealing with *strong quasi-reducibility* in Section 3.2, one extending it to general quasi-reduciblity in Section 3.3 and a third, applying the algorithm to higher-order applicative term rewriting in Section 3.4. The main theorems are discussed in Chapter 4, where Section 4.1 discusses the termination argument and Section 4.2 the soundness of the algorithm.

The main results of the paper are Theorem 1 detailing the termination of the quasi-reducibility decision procedure and Theorem 3 detailing the soundness of the algorithm.

Further research could explore quasi-reducibility and related notions in higher-order rewriting while also maintaining some restrictions to keep the algorithms decidable. Moreover, the complexity of such algorithms is also of importance, as can be seen in a recent paper by Thiemann in IWC 2025 [12].

Bibliography

[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] 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`.

[3] 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`.

[4] 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`.

[5] Azeddine Lazrek, Pierre Lescanne, and Jean-Jacques Thiel. “Tools for proving inductive equalities, relative completeness, and ω-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`.

[6] GHC Team. *The Glasgow Haskell Compiler*. `https://www.haskell.org/ghc/`. Accessed: 2025-06-01.

[7] Paolo Capriotti and Ambrus Kaposi. "Free Applicative Functors". *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), pp. 2–30. ISSN: 2075-2180. DOI: `10.4204/eptcs.153.2`. URL: `http://dx.doi.org/10.4204/EPTCS.153.2`.

[8] Daniel Koves. "Literature Study on Algorithms for Pattern Completeness". Unpublished manuscript. 2025.

[9] 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`.

[10] 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`.

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

[12] IWC 2025. *14th International Workshop on Confluence.* `https://iwc2025.github.io`. Accessed: 2025-08-26.