Static analysis of pattern-free properties

Horatiu Cirstea
Université de Lorraine – LORIA
Nancy, France
Horatiu.Cirstea@loria.fr

Pierre Lermusiaux
Université de Lorraine – LORIA
Nancy, France
Pierre.Lermusiaux@loria.fr

Pierre-Etienne Moreau
Université de Lorraine – LORIA
Nancy, France
Pierre-Etienne.Moreau@loria.fr

ABSTRACT

Rewriting is a widely established formalism with major applications in computer science. It is indeed a staple of many formal verification applications as it is especially well suited to describe program semantics and transformations. In particular, constructor based term rewriting systems are generally used to illustrate the behaviour of functional programs.

In the context of formal verification, it is often necessary to characterize the shape of the reducts of such rewrite systems and, in a typed context, the underlying type system provides syntactic guarantees on the form of these terms by exhibiting, among others, the constructor symbols that they can contain. On the other hand, when performing (program) transformations we often want to eliminate some symbols and, more generally, to ensure that some patterns are absent from the result of the transformation.

We propose in this paper an approach to statically verify the absence of specified patterns from the reachable terms of constructor based term rewriting systems. The proposed approach consists in annotating the function symbols with a set of profiles outlining pre- and post-conditions that must be verified by the rewrite relation, and using a rewrite based method to statically verify that the rewrite system is indeed consistent with the respective annotations.

CCS CONCEPTS

• **Theory of computation → Logic and verification**.

KEYWORDS

Rewriting, Pattern-matching, Pattern semantics, Static analysis

ACM Reference Format:
Horatiu Cirstea, Pierre Lermusiaux, and Pierre-Etienne Moreau. 2021. Static analysis of pattern-free properties. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021), September 6–8, 2021, Tallinn, Estonia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3479394.3479404

ACKNOWLEDGMENTS

This work is partially supported by the Agence Nationale de la Recherche under project ANR-16-CE25-0007 FORMEDICIS (https://anr.fr/Projet-ANR-16-CE25-0007).

1 INTRODUCTION

Rewriting is a robust formalism, which not only provides a useful framework for formal verification but also a broad expressive power and is thus especially well suited in the context of declarative programming. In particular, these properties advocate it as a comprehensive approach used to describe program semantics [26] and program transformations [6, 22]. Languages implementing the underlying notions of pattern matching and rewrite rules include well known functional languages as well as rule based languages, like Maude [11], Stratego [31], or Tom [4].

Giving a characterization of the set of terms reachable through a transformation is a problem that has been extensively studied for its major implications in formal verification. For example, infinite-state systems can be modeled using term rewriting systems or tree transducers, and multiple approaches, ranging from model checking ones [8, 21] to tree automata completion [14] can then be used to verify, by reachability analysis, some forms of correctness properties.

In the context of program transformation, a similar approach [9] was introduced to verify the correctness of a transformation with respect to the target language and, more precisely, to guarantee that specific constructor symbols, or patterns, are absent from the programs obtained by transformation. This approach relies on a system of annotations of the function symbols by the patterns to be eliminated.

For instance, let us consider lists of expressions build out of (wrapped) integers and lists:

$$\begin{array}{rcl} Expr & = & int(Int) \ \mid \ lst(List) \\ List & = & nil \ \mid \ cons(Expr, List) \end{array}$$

and a transformation which flattens list expressions implemented by the functions $flatten : List \mapsto List$ and $concat : List * List \mapsto List$ defined using the rewriting system $\mathcal{R}$:

$$\begin{cases} flatten(nil) & \twoheadrightarrow & nil \\ flatten(cons(int(n), l)) & \twoheadrightarrow & cons(int(n), flatten(l)) \\ flatten(cons(lst(l), l')) & \twoheadrightarrow & flatten(concat(l, l')) \\ concat(cons(e, l), l') & \twoheadrightarrow & cons(e, concat(l, l')) \\ concat(nil, l) & \twoheadrightarrow & l \end{cases}$$

To statically ensure that the result of the transformation contains no nested lists we could of course introduce new types $List'$ and $Expr'$ describing such lists and change accordingly the types of the functions. This is quite intrusive and particularly tedious when the differences between the source and the target languages only concern a few symbols but induce nevertheless significant type extensions.

The approach proposed in [9] allows the verification of the property by simply annotating the function symbol *flatten* with the corresponding (anti-)pattern $p := cons(lst(l_1), l_2)$ and checking that the rewriting system is consistent with the annotation, and thus, that the normal forms are flat lists. The method relies on an over-approximation of the potential results obtained by reduction and thus, it may lead to some false negatives. For example, the system $\mathcal{R}'$ obtained from $\mathcal{R}$ by replacing the third rule with

$$\mathit{flatten}(\mathit{cons}(\mathit{lst}(l), l')) \twoheadrightarrow \mathit{concat}(\mathit{flatten}(l), \mathit{flatten}(l'))$$

while still producing only flat lists, could not be verified with this method.

We propose in this paper a more elaborate annotation system which declares for each function not only a single (anti-)pattern specifying the post-conditions on the expected outcome but also the pre-conditions guaranteeing the form of the results. This allows for a substantially more precise description of the expected behaviour of the reduction associated to each function symbol, with the aim of reducing significantly the presence of false negatives. For each function we have one or several profiles and, for example, the *concat* function in the rewriting system $\mathcal{R}'$ can be annotated by the profile $p{*}p \mapsto p$ to indicate that the concatenation of two flat lists is a flat list. The new approach relies on an inference method to characterize the substitutions consistent with the pre-conditions and a verification method guaranteeing that the application of these substitutions is consistent with the post-condition. The method applies to Constructor Based Term Rewriting Systems (CBTRS), which are a common formal representation of functional programs relying on pattern matching.

Contributions. We thus provide a method to verify that, given a set of functions whose behaviours are described by a given CBTRS, and a set of annotations for the corresponding function symbols, the normal forms potentially obtained by rewriting are consistent with the annotations. More precisely, given a function symbol f annotated with a profile $p_1{*}\dots{*}p_n \mapsto p$, we can verify that for any values $v_1, \dots, v_n$ containing no subterm matched by $p_1, \dots, p_n$ respectively, the values potentially obtained by reducing the term $f(v_1, \dots, v_n)$ with the CBTRS defining f contain no subterm matched by p. Note that, while this result is ultimately reliant on the termination of the considered CBTRS, the proposed method is based on a notion of semantics preservation that does not rely on the termination or confluence of the CBTRS when this is linear. For non right-linear CBTRSs, we propose an adjustment of the method that assumes a strict reduction strategy and applies effectively only for confluent CBTRSs.

First, we introduce basic notions and notations for term rewriting and pattern-matching. We revisit in Section 3 the notions of pattern-freeness and term semantics needed for our extended profiles. In Section 4 we describe the inference and checking methods and in Section 5 we explain how the overall method adapts to some cases of non-linearity. We finally present some related work and conclude.

2 GENERAL NOTIONS AND NOTATIONS

We present in this section the basic notions and notations used in this paper. More details on term rewriting systems can be found in [3, 28]. The extended patterns related notions introduced in [9, 10] are adapted here to cope with non-linear patterns.

2.1 Constructor Based Term Rewriting System

A *many-sorted signature* $\Sigma = (\mathcal{S}, \mathcal{F})$, consists of a set of sorts $\mathcal{S}$ and a set of symbols $\mathcal{F}$. We distinguish constructor symbols from function symbols by partitioning the alphabet $\mathcal{F}$ into $\mathcal{D}$, the set of *defined symbols*, and $\mathcal{C}$ the set of *constructors*: $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$. A symbol f with *domain* $s_1{*}\cdots{*}s_n \in \mathcal{S}^*$ and *co-domain* $s \in \mathcal{S}$ is written $f : s_1{*}\cdots{*}s_n \mapsto s$; we may write f_s to indicate explicitly the co-domain. We denote by $\mathcal{C}_s$, resp. $\mathcal{D}_s$, the set of constructors, resp. defined symbols, with co-domain s. Variables are also sorted and we write x_s to indicate that the variable x has sort s. The set $\mathcal{X}_s$ denotes a set of variables of sort s and $\mathcal{X} = \bigcup_{s\in\mathcal{S}} \mathcal{X}_s$ is the set of sorted variables. In what follows, we explicitly indicate the sort for a variable, or for a set, only if it cannot be implicitly inferred or if we want to emphasize this information.

The set of terms of sort $s \in \mathcal{S}$, denoted $\mathcal{T}_s(\mathcal{F}, \mathcal{X})$ is the smallest set containing $\mathcal{X}_s$ and such that $f(t_1, \dots, t_n)$ is in $\mathcal{T}_s(\mathcal{F}, \mathcal{X})$ whenever $f : s_1{*}\cdots{*}s_n \mapsto s$ and $t_i \in \mathcal{T}_{s_i}(\mathcal{F}, \mathcal{X})$ for $i \in [1, n]$. We write $t : s$ to explicitly indicate that the term t is of sort s, *i.e.* when $t \in \mathcal{T}_s(\mathcal{F}, \mathcal{X})$. The set of *sorted terms* is defined as $\mathcal{T}(\mathcal{F}, \mathcal{X}) = \bigcup_{s\in\mathcal{S}} \mathcal{T}_s(\mathcal{F}, \mathcal{X})$. The set of variables occurring in $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground* term. $\mathcal{T}_s(\mathcal{F})$ denotes the set of all ground first-order terms of sort s and $\mathcal{T}(\mathcal{F})$ denotes the set of all ground first-order terms, while members of $\mathcal{T}(\mathcal{C})$ are called *values*. A *linear* term is a term where every variable occurs at most once. The terms in $\mathcal{T}(\mathcal{C}, \mathcal{X})$ are called *constructor patterns*.

A *position* of a term t is a finite sequence of positive integers describing the path from the root of t to the root of the subterm at that position. The empty sequence representing the root position is denoted by ε. $t_{|\omega}$ denotes the subterm of t at position ω, and $t\,[r]_\omega$ the term t with the subterm at position ω replaced by r. We note $\mathcal{P}os(t)$ the set of positions of t.

We call *substitution* any mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ which is the identity except over a finite set of variables $\mathcal{D}om(\sigma)$ called its *domain*. A substitution σ extends as expected to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We call *value substitution* any substitution σ mapping variables to values, *i.e.* such that $\sigma(x) \in \mathcal{T}(\mathcal{C})$ for any $x \in \mathcal{D}om(\sigma)$. Sorted substitutions are such that if $x : s$ then $\sigma(x) : s$. Note that for any such sorted substitution σ, $t : s$ iff $\sigma(t) : s$. In what follows we will only consider such sorted substitutions.

A *constructor rewrite rule* (over Σ) is a pair of terms $(\varphi(l_1, \dots, l_n), r) \in \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \times \mathcal{T}_s(\mathcal{F}, \mathcal{X})$ (usually denoted $\varphi(l_1, \dots, l_n) \twoheadrightarrow r$) with $s \in \mathcal{S}$, $\varphi \in \mathcal{D}$, $l_1, \dots, l_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ and such that $l := \varphi(l_1, \dots, l_n)$ is linear and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A *constructor based term rewriting system* (CBTRS) is a set of constructor rewrite rules $\mathcal{R}$ inducing a *rewrite relation* over $\mathcal{T}(\mathcal{F})$, denoted by $\Longrightarrow_{\mathcal{R}}$ and such that $t \Longrightarrow_{\mathcal{R}} t'$ iff there exists $l \twoheadrightarrow r \in \mathcal{R}$, $\omega \in \mathcal{P}os(t)$ and a substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t\,[\sigma(r)]_\omega$. We note $\longrightarrow_R$ the rewrite relation induced by $\mathcal{R}$ with a strict reduction strategy, *i.e.* such that $t \longrightarrow_{\mathcal{R}} t'$ iff there exists $l \twoheadrightarrow r \in \mathcal{R}$, $\omega \in \mathcal{P}os(t)$ and a value substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t\,[\sigma(r)]_\omega$. $\Longrightarrow^*_{\mathcal{R}}$, resp. $\longrightarrow^*_{\mathcal{R}}$, denotes the reflexive and transitive closure of $\Longrightarrow_{\mathcal{R}}$, resp. $\longrightarrow_{\mathcal{R}}$.

2.2 Extended Patterns and Ground Semantics

Starting from constructor patterns, we adapt the notion of *extended pattern* first introduced in [10]:

Definition 2.1 (Extended patterns). *Given a set of variables $\mathcal{X}$ and a signature $\Sigma = (\mathcal{S}, \mathcal{D} \uplus \mathcal{C})$ the set $\mathcal{P}(\mathcal{C}, \mathcal{X})$ of* extended patterns *is defined as follows:*

$$p, q \quad := \quad x \mid c(q_1, \dots, q_n) \mid p_1 + p_2 \mid p_1 \setminus p_2 \mid p_1 \times p_2 \mid \bot$$

with $x \in \mathcal{X}, p, p_1, p_2 : s$ for some $s \in \mathcal{S}, c : s_1 * \cdots * s_n \mapsto s \in \mathcal{C}$ and $\forall i \in [1, n], q_i : s_i$. We assume $+, \setminus, \times, \bot \notin \mathcal{D} \uplus \mathcal{C}$. The set of variables occurring in an extended pattern p is denoted $\mathcal{V}ar(p)$.

An extended pattern $p \in \mathcal{P}(\mathcal{C}, \mathcal{X})$ matches a value $v \in \mathcal{T}(\mathcal{C})$, denoted $p \lll v$, iff there exists a substitution σ such that $p \overset{\sigma}{\lll} v$ with:

$$\begin{array}{rcl} x \overset{\sigma}{\lll} v & & \text{iff } v = \sigma(x) \\ c(p_1, \dots, p_n) \overset{\sigma}{\lll} c(v_1, \dots, v_n) & & \text{iff } \wedge_{i=1}^{n} p_i \overset{\sigma}{\lll} v_i, \text{ for } c \in \mathcal{C} \\ p_1 + p_2 \overset{\sigma}{\lll} v & & \text{iff } p_1 \overset{\sigma}{\lll} v \ \vee \ p_2 \overset{\sigma}{\lll} v \\ p_1 \setminus p_2 \overset{\sigma}{\lll} v & & \text{iff } p_1 \overset{\sigma}{\lll} v \ \wedge \ p_2 \not\lll v \\ p_1 \times p_2 \overset{\sigma}{\lll} v & & \text{iff } p_1 \overset{\sigma}{\lll} v \ \wedge \ p_2 \overset{\sigma}{\lll} v \end{array}$$

Note that $\bot \not\lll v$ for any v.

A pattern $p_1 + p_2$ matches any term matched by one of its components while a pattern $p_1 \times p_2$ matches any term matched by both its components. The relative complement of p_2 *w.r.t.* p_1, $p_1 \setminus p_2$, matches all terms matched by p_1 but those matched by p_2.

The relation holds not only for patterns against values but also for general terms containing no complement patterns; it is not stable by reduction in presence of complements and a more elaborate definition should be considered to tackle the problem. The notions used in what follows only involve matchings of extended patterns against values but the underlying matching of the constructor patterns in CBTRS reductions is done against general terms.

The *ground semantics* of an extended pattern is the set of values matching the pattern [10] and thus, the ground semantics of a constructor pattern is the set of all its value instances:

$$[\![p]\!] = \{v \in \mathcal{T}(\mathcal{C}) \mid p \lll v\}$$

Given an extended pattern p, some of its variables are not significant for the matching, *i.e.* for all $v \in [\![p]\!]$, they are not in the domain of σ in the relation $p \overset{\sigma}{\lll} v$ introduced in Definition 2.1.

The set $\mathcal{MV}(p)$ of *matchable variables* of a pattern p is defined as follows:

$$\begin{array}{rcl} \mathcal{MV}(x) & = & \{x\}, \text{for all } x \in \mathcal{X} \\ \mathcal{MV}(c(p_1, \dots, p_n)) & = & \mathcal{MV}(p_1) \cup \cdots \cup \mathcal{MV}(p_n), \text{for all } c \in \mathcal{C} \\ \mathcal{MV}(p_1 + p_2) & = & \mathcal{MV}(p_1) \cup \mathcal{MV}(p_2) \\ \mathcal{MV}(p_1 \times p_2) & = & \mathcal{MV}(p_1) \cup \mathcal{MV}(p_2) \\ \mathcal{MV}(p_1 \setminus p_2) & = & \mathcal{MV}(p_1) \\ \mathcal{MV}(\bot) & = & \emptyset \end{array}$$

The variables of p which are not matchable are *free*: $\mathcal{FV}(p) = \mathcal{V}ar(p) \setminus \mathcal{MV}(p)$. In what follows we consider that matchable and free variables have different names.

A pattern $c(p_1, \dots, p_n)$ is linear if each p_i, $i \in [1, n]$, is linear and $\forall 1 \leq i < j \leq n$, $\mathcal{MV}(p_i) \cap \mathcal{MV}(p_j) = \emptyset$. The pattern $p_1 + p_2$, resp. $p_1 \setminus p_2$, is linear if each of p_1 and p_2 is linear. This corresponds to the fact that p_1 and p_2 represent independent alternatives and thus, that their variables are unrelated *w.r.t.* pattern semantics. For example, the terms $h(x) + g(x)$ and $h(x) + g(y)$ both represent all terms rooted by h or g. The pattern $p_1 \times p_2$ is linear if each of p_1 and p_2 is linear and $\mathcal{MV}(p_1) \cap \mathcal{MV}(p_2) = \emptyset$. This corresponds to the fact that p_1 and p_2 should be matched simultaneously against the corresponding value.

In this paper we consider that all conjunctions $p_1 \times p_2$ are such that $\mathcal{MV}(p_1) \cap \mathcal{MV}(p_2) = \emptyset$ and, in this case, for any patterns p, q we have $[\![p \times q]\!] = [\![p]\!] \cap [\![q]\!]$. This property is crucial for checking pattern-freeness with our approach.

If a pattern contains no $\bot$ (except at the root position), no $\times$ and no $\setminus$ it is called *additive*.

3 ANNOTATED TERMS AND SEMANTICS PRESERVING CBTRS

The approach proposed in [9] used defined symbol annotations to indicate the absence of the specified patterns from the reducts of a term headed by the respective symbols. There was nevertheless no indication on the shape of the arguments of these defined symbols and this lack of precision could lead to some false negatives. We consider here a more elaborate profile for the defined symbols to perform a finer grained analysis of the potential normal forms.

Although no restriction is imposed on the analysed CBTRS, the method described in what follows is really interesting when the system is at least weakly normalizing in which case we guarantee the absence of specified patterns from the normal forms.

3.1 Pattern-free Terms and their Semantics

Every defined symbol with a sort profile $f : s_1 * \cdots * s_n \mapsto s \in \mathcal{D}$ is associated with one or several pattern profiles $p_1 * \cdots * p_n \mapsto p$, with $p_1, \dots, p_n, p$ linear additive patterns. Intuitively, such a pattern profile indicates that the normal form of a ground term of the form $f(t_1, \dots, t_n)$, when it exists, is a p-free value if the terms $t_1, \dots, t_n$ can be reduced only to values that are respectively p_1-, …, p_n-free. We consider that every defined symbol $f^{!\mathcal{P}}$ is annotated with the set $\mathcal{P}$ of its profiles; when $\mathcal{P}$ is empty the annotation is omitted.

Given the example from the introduction, we could consider for the symbol *concat* the profile $p{*}p \mapsto p$, with $p := cons(lst(l1), l2)$, to indicate that the concatenation of two flat lists is a flat list. We could also use a second profile $q{*}q \mapsto q$, with $q := cons(e, l)$, to indicate that the concatenation of two empty lists is an empty list.

We consider terms whose instances and reducts can't be matched by a specified pattern:

Definition 3.1 (Pattern-free terms). *Given an additive pattern p,*

- *a value $v \in \mathcal{T}(\mathcal{C})$ is p-free iff $\forall \omega \in \mathcal{P}os(v), p \not\lll v_{|\omega}$;*
- *a pattern $u \in \mathcal{T}(\mathcal{C}, \mathcal{X}) \setminus \mathcal{T}(\mathcal{C})$ is p-free iff $\forall \sigma$ s.t. $\sigma(u) \in \mathcal{T}(\mathcal{C})$, $\sigma(u)$ is p-free;*
- *a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \setminus \mathcal{T}(\mathcal{C}, \mathcal{X})$ is p-free iff $\forall \omega \in \mathcal{P}os(t)$ s.t. $t_{|\omega} = f_s^{!\mathcal{P}}(t_1, \dots, t_n)$ with $f_s^{!\mathcal{P}} \in \mathcal{D}_s$, the term $t\,[v]_\omega$ is p-free for all q-free value $v \in \mathcal{T}_s(\mathcal{C})$ where $q = \sum_{q' \in \mathcal{Q}} q'$, $\mathcal{Q} = \{r \mid \exists r_1 * \cdots * r_n \mapsto r \in \mathcal{P}$ s.t. $\forall i \in [1, n], t_i$ is r_i-free$\}$.*

A value is p-free if and only if p matches no subterm of the value. For constructor patterns, verifying a pattern-free property comes to verifying the property for all the ground instances of the term. Finally, a general term is p-free if and only if for all subterms u

headed by a defined symbol $f \in \mathcal{D}_s$ replacing all instances of u in t by any q-free value of sort s, with q a pattern built by taking into account profiles consistent with u, results in a p-free term. Intuitively, this corresponds to considering an over-approximation of the set of potential normal forms of that subterm, and therefore of the whole term.

Note that $\mathcal{Q}$ could be empty (for example, when $\mathcal{P}$ is empty) in which case $q = \perp$ and in this case v can be any value. We can also remark that all terms are $\perp$-free.

Example 3.1. *We consider the signature $\Sigma = (\mathcal{S}, \mathcal{F})$ with $\mathcal{S} = \{Expr, List\}$ and $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{C} = \{int : Int \mapsto Expr,\ lst : List \mapsto Expr,\ nil : List,\ cons : Expr{*}List \mapsto List\}$, and $\mathcal{D} = \{flatten^{!\mathcal{P}_1} : List \mapsto List,\ concat^{!\mathcal{P}_2} : List{*}List \mapsto List\}$.*

When we take $\mathcal{P}_1 = \{\perp \mapsto p_1\}$, with $p_1 := cons(lst(l_1), l_2)$, and $\mathcal{P}_2 = \emptyset$, the term $flatten(l)$ is p_1-free since $q = p_1$ in Definition 3.1. On the other hand, the term $concat(l, l')$ is not p_1-free since $q = \perp$ and there obviously exists a $\perp$-free value which is not p_1-free. Similarly, neither $concat(nil, nil)$ nor $concat(flatten(l), flatten(l'))$ is p_1-free. If we change the annotation of concat to $\mathcal{P}_2 = \{p_1{*}p_1 \mapsto p_1, p_2{*}p_2 \mapsto p_2\}$, with p_1 as before and $p_2 := cons(e, l)$, we have:

- *$concat(l, l')$ is neither p_1-free nor p_2-free;*
- *$concat(flatten(l), flatten(l'))$ is p_1-free (since $q = p_1$ in Definition 3.1), but not p_2-free;*
- *$concat(nil, nil)$ is p_1-free and p_2-free, i.e. $(p_1 + p_2)$-free, (since $q = p_1 + p_2$ in Definition 3.1).*

The pattern-free properties in the above example have been checked by verifying manually the (pattern matching) conditions required by the pattern-free definition. We propose in Section 4.3 a method relying on the notion of ground semantics to verify automatically pattern-free properties. For this, we should first extend the notion of ground semantics to take into account all terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$:

Definition 3.2 (Generalized ground semantics). *Let $p \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,*

- *$\llbracket p \rrbracket = \{v \in \mathcal{T}(\mathcal{C}) \mid p \ll v\} = \{\sigma(p) \mid \sigma(p) \in \mathcal{T}(\mathcal{C})\}$;*
- *$\llbracket t \rrbracket = \bigcup_\omega \bigcup_v \llbracket t\,[v]_\omega \rrbracket$ with $\omega \in \mathcal{P}os(t)$ such that $t_{|\omega} = f_s^{!\mathcal{P}}(t_1, \ldots, t_n), f_s^{!\mathcal{P}} \in \mathcal{D}_s$, and $v \in \mathcal{T}_s(\mathcal{C})$ q-free, where $q = \sum_{q' \in \mathcal{Q}} q'$, $\mathcal{Q} = \{r \mid \exists r_1{*}\cdots{*}r_n \mapsto r \in \mathcal{P}$ s.t. $\forall i \in [1, n], t_i$ is r_i-free$\}$.*

For constructor patterns, the definition of ground semantics is unchanged. The ground semantics of a term containing defined symbols represents the union of the ground semantics of the terms obtained by over-approximating all the possible reductions of the subterms headed by such defined symbols as indicated by the annotation of the respective symbols. The over-approximation is identical to the one considered in the pattern-free property, and directly related to the chosen annotations. As such, the more precise the profiles describe the behaviour of the reduction of a defined symbol, the more precise the approximation is.

A term is p-free if all the values in its semantics are p-free:

Proposition 3.1 (Pattern-free vs Ground Semantics). *Given $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and p an additive pattern, t is p-free iff $\forall v \in \llbracket t \rrbracket, v$ is p-free.*

The notion of ground semantics can be extended to the more general notion of *deep semantics* which is closed by the subterm relation: $\{\!\![t]\!\!\} = \{u_{|\omega} \mid u \in \llbracket t \rrbracket, \omega \in \mathcal{P}os(u)\}$. Note that this notion introduces no additional over-approximation, all over-approximations being situated at the level of ground semantics. Then, checking that t is p-free simply consists in checking that the intersection between the deep semantics of t and the ground semantics of p is empty:

Proposition 3.2 (Pattern-free vs Deep Semantics). *Given $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and p an additive pattern, t is p-free iff $\{\!\![t]\!\!\} \cap \llbracket p \rrbracket = \emptyset$.*

We will see in Section 4.3 how the above intersection can be computed and we first look at how pattern-freeness interacts with term rewriting.

3.2 Semantics preserving CBTRS

Generalized ground semantics rely on the symbol annotations and assume thus a specific shape for the potential normal forms of reducible terms. This assumption should be checked by verifying that the CBTRSs defining the annotated symbols are consistent with these annotations, *i.e.* check that the semantics is preserved by reduction.

Definition 3.3 (Semantics preservation). *We say that a rewrite rule $ls \rightarrowtriangle rs$ is* semantics preserving *iff, for all substitution σ, we have $\llbracket \sigma(rs) \rrbracket \subseteq \llbracket \sigma(ls) \rrbracket$. We say that a rewrite rule $ls \rightarrowtriangle rs$ is* strictly semantics preserving *iff, for all* value *substitution σ, we have $\llbracket \sigma(rs) \rrbracket \subseteq \llbracket \sigma(ls) \rrbracket$.*

A CBTRS is (strictly) semantics preserving iff all its rewrite rules are.

Semantics preservation carries over to the induced rewrite relation:

Proposition 3.3. *Given a semantics preserving CBTRS $\mathcal{R}$, we have*

$$\forall t, v \in \mathcal{T}(\mathcal{F}),\ \text{if } t \Longrightarrow_{\mathcal{R}}^{*} v,\ \text{then } \llbracket v \rrbracket \subseteq \llbracket t \rrbracket.$$

Together with Proposition 3.1 this guarantees that the rewrite relation induced by a semantics preserving CBTRS preserves pattern-free properties. Thus, the semantics preservation allows the characterization of the potential normal forms for a semantics preserving CBTRS and in what follows we give sufficient conditions guaranteeing this latter property and provide a method for automatically checking these conditions.

Definition 3.4 (Profile satisfaction). *Given a constructor rewrite rule $f(ls_1, \ldots, ls_n) \rightarrowtriangle rs$ and a profile $\pi = p_1{*}\cdots{*}p_n \mapsto p$, we say that the rule satisfies, respectively strictly satisfies, the profile π iff, for all substitution, respectively value substitution, σ with $\mathcal{D}om(\sigma) = \mathcal{V}ar(f(ls_1, \ldots, ls_n))$,*

$$\sigma(ls_i) \text{ is } p_i\text{-free for all } i \in [1, n] \quad \Longrightarrow \quad \sigma(rs) \text{ is } p\text{-free}$$

Profile satisfaction is a necessary and sufficient condition for semantics preservation:

Proposition 3.4 (Semantics preservation). *A constructor rewrite rule $f^{!\mathcal{P}}(ls_1, \ldots, ls_n) \rightarrowtriangle rs$ is semantics preserving, respectively strictly semantics preserving, if and only if it satisfies, respectively strictly satisfies, all profiles of $\mathcal{P}$.*

A rewrite rule is thus semantics preserving if whenever the specified profiles are verified by the left-hand side they are also verified by the right-hand side. To perform this verification we first infer the shapes of the terms that could be used to instantiate the variables in the left-hand side such that a given profile of the head symbol is verified and then, check that when replacing accordingly the variables in the right-hand side we respect the profile.

Example 3.2. *We consider the signature from Example 3.1 and examine the rewrite rule $concat(cons(e,l),l') \twoheadrightarrow cons(e, concat(l,l'))$ from the CBTRS presented in the introduction.*

Given the substitution $\sigma_1 = \{e \mapsto int(5), l \mapsto nil, l' \mapsto nil\}$ we can easily see that $\sigma_1(cons(e,l)) = cons(int(5), nil)$ and $\sigma_1(l') = nil$ are p_1-free and thus, we should check that $\sigma_1(cons(e, concat(l,l'))) = cons(int(5), concat(nil, nil))$ is p_1-free. As we have seen in Example 3.1, $concat(nil, nil)$ is p_1-free and, since we have $cons(lst(l_1), l_2) \not\ll cons(int(5), v)$ for all value v, $\sigma_1(cons(e, concat(l,l'))) = cons(int(5), concat(nil, nil))$ is also p_1-free (Definition 3.1).

For the substitution $\sigma_2 = \{e \mapsto lst(cons(int(5), nil)), l \mapsto nil, l' \mapsto nil\}$, while $\sigma_2(l') = nil$ is clearly p_1-free, $\sigma_2(cons(e,l)) = cons(lst(cons(int(5), nil)), nil)$ is not, and thus there is no need to check that $\sigma_2(cons(e, concat(l,l')))$ is p_1-free when verifying the profile satisfaction of the rule.

4 VERIFYING SEMANTICS PRESERVATION

The first step towards the verification of the profile satisfaction of a rule $f^{!\mathcal{P}}(ls_1, \ldots, ls_n) \twoheadrightarrow rs$ consists in characterizing, for each profile $p_1 * \ldots * p_n \mapsto p \in \mathcal{P}$, the substitutions σ such that $\sigma(ls_i)$ is p_i-free or equivalently, inferring the pattern-freeness constraints on the shape of the values that could be substituted for the variables of ls_i so that all the corresponding instances of ls_i are p_i-free. These constraints are naturally expressed in terms of extended patterns using annotated variables and an aliasing mechanism.

4.1 Annotated and alias variables

We consider a set $\mathcal{X}^a$ of variables annotated by linear additive patterns, and extend the matching relation to such variables:

$$x^{-p} \overset{\sigma}{\ll} v \quad \text{iff } v = \sigma(x^{-p}) \text{ and } v \text{ is } p\text{-free}$$

Note that the ground semantics of a plain variable x_s is the set of all possible values of a given sort: $[\![x_s]\!] = \mathcal{T}_s(\mathcal{C})$, while the ground semantics of an annotated variable x_s^{-p} is the set of all possible p-free values $[\![x_s^{-p}]\!] = \{v \in \mathcal{T}_s(\mathcal{C}) \mid v \text{ } p\text{-free}\}$. Since $x_s^{-\perp}$ has the same semantics as x_s we use indistinguishably x_s and $x_s^{-\perp}$.

Extended patterns involve now annotated variables and patterns aliased by variables:

$$p, q \ := \ x \mid c(q_1, \ldots, q_n) \mid p_1 + p_2 \mid p_1 \setminus p_2 \mid p_1 \times p_2 \mid \perp \mid y @ p$$

with $x \in \mathcal{X}_s^a, y \in \mathcal{X}_s, p, p_1, p_2 : s$ for some $s \in \mathcal{S}, \forall i \in [1,n], q_i : s_i$ and $c : s_1 * \cdots * s_n \mapsto s \in \mathcal{C}$. @ has a higher priority than $\setminus$ which has a higher priority than + and $\times$.

Pattern matching and matchable variables for aliased patterns are defined as for conjunctions: $x @ p \overset{\sigma}{\ll} v$ iff $x \times p \overset{\sigma}{\ll} v$, and $\mathcal{MV}(x @ p) = \{x\} \cup \mathcal{MV}(p)$. In what follows we consider only aliased patterns $x@p$ such that $x \notin \mathcal{MV}(p) = \emptyset$, for which $[\![x@p]\!] = [\![p]\!]$.

We consider from now on extended patterns in $\mathcal{P}(\mathcal{C}, \mathcal{X}^a)$ and substitutions which map variables from $\mathcal{X}$ to $\mathcal{P}(\mathcal{C}, \mathcal{X}^a)$. We call *symbolic* the patterns in $\mathcal{T}(\mathcal{C}, \mathcal{X}^a)$ and *regular* patterns that contain only variables of the form $x^{-\perp}$. We call *quasi-additive* patterns that contain no $\times$ and only contain $\setminus$ with the pattern on the left being a variable and the pattern on the right being a regular additive pattern. And we call *quasi-symbolic* a quasi-additive pattern that contains no $\perp$ and only contains + in the additive pattern on the right of a $\setminus$. Additive patterns are defined as before and thus, they contain no aliases. The annotation of an alias variable is always $\perp$ and, for readability purposes, it is omitted in what follows. Finally, given a quasi-symbolic pattern t, we consider the set of its aliased variables $\mathcal{A}t\mathcal{V}ar(t) = \{x \in \mathcal{V}ar(t) \mid \exists \omega \in \mathcal{P}os(t), u \in \mathcal{P}(\mathcal{C}, \mathcal{X}^a) \ . \ t_{|\omega} = x @ u\}$ and for any $x \in \mathcal{A}t\mathcal{V}ar(t)$ we note $t_{@x} = \prod_{q \in Q_{@x}} q$ with $Q_{@x} = \{q \mid \exists \omega \in \mathcal{P}os(t), t_{|\omega} = x @ q\}$.

For example, the pattern $cons(e @ (x \setminus lst(l_1)) + lst(nil), l @ y)$ is quasi-additive but not quasi-symbolic, while $t = cons(e @ (x \setminus (lst(l_1) + int(0))), l @ y)$ and $u = cons(lst(l @ cons(int(n), l_1)), l @ cons(e, nil))$ are quasi-symbolic. Moreover, we have $t_{@e} = x \setminus (lst(l_1) + int(0))$ and $u_{@l} = cons(int(n), l_1) \times cons(e, nil)$.

We assume a set $\mathcal{U} = \{\cdot_1, \ldots, \cdot_n\}$ of suitable symbols for n-tuples (the cardinality of $\mathcal{U}$ is the maximum arity of the symbols in $\mathcal{D}$), and for simplicity an n-tuple $\cdot_n(p_1, \ldots, p_n)$ is denoted $(\!| p_1, \ldots, p_n |\!)$. Tuple symbols behave as constructor symbols *w.r.t.* all definitions.

Definition 4.1 (Aliased versions). *A term τ is an* aliased version *of a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ iff for all $\omega \in \mathcal{P}os(t)$ s.t. $t_{|\omega} = x \in \mathcal{V}ar(t)$ we have $\tau_{|\omega} = x @ q$ for some quasi-symbolic pattern q with $\mathcal{MV}(q) \cap \mathcal{V}ar(t) = \emptyset$, and for all distinct variables $x, y \in \mathcal{V}ar(t)$ we have $\mathcal{MV}(\tau_{@x}) \cap \mathcal{MV}(\tau_{@y}) = \emptyset$.*

$\sigma_t^{@\tau}$ denotes the substitution corresponding to the mapping $\{x \mapsto x @ \tau_{@x} \mid x \in \mathcal{V}ar(t)\}$.

For example, $cons(e @ (x \setminus lst(l_1)), l @ y)$ is an aliased version of $cons(e,l)$, constraining the variables e, and l, to terms matching $x \setminus lst(l_2)$ (*i.e.* not lists), and y (*i.e.* any term), respectively.

4.2 Inferring the shape of variables

To characterize substitutions σ such that $\sigma(ls_i)$ is p_i-free we note that, according to Proposition 3.1 and to the definition of annotated variables, $\sigma(ls_i)$ is p_i-free iff $[\![\sigma(ls_i)]\!] \subseteq [\![x_{s_i}^{-p_i}]\!]$. Since, ls_i is a constructor pattern whose semantics is the set of all its instances, $[\![\sigma(ls_i)]\!] \subseteq [\![ls_i]\!]$. Thus, $\sigma(ls_i)$ is p_i-free iff $[\![\sigma(ls_i)]\!] \subseteq [\![ls_i]\!] \cap [\![x_{s_i}^{-p_i}]\!] = [\![ls_i \times x_{s_i}^{-p_i}]\!]$ (with $x_{s_i}^{-p_i}$ a fresh variable).

We express each extended pattern $ls_i \times x_{s_i}^{-p_i}$ in terms of a sum of aliased versions of ls_i, $[\![ls_i \times x_{s_i}^{-p_i}]\!] = [\![\lambda_i^1 + \lambda_i^2 + \cdots + \lambda_i^m]\!]$ (with m and $\lambda_i^k, k \in [1 \ldots m]$ depending, as we will see below, on ls_i and $x_{s_i}^{-p_i}$), such that for all σ with $\sigma(ls_i)$ p_i-free, $[\![\sigma(ls_i)]\!] \subseteq [\![\lambda_i^k]\!]$ for some $k \in [1 \ldots m]$. The substitution can then be characterized by specifying the constraints on the pattern's variables:

Lemma 4.1. *Given λ, an aliased version of a constructor pattern l, we have: $\forall \sigma \ ([\![\sigma(l)]\!] \subseteq [\![\lambda]\!] \iff \forall x \in \mathcal{V}ar(l), [\![\sigma(x)]\!] \subseteq [\![\lambda_{@x}]\!])$*

Once the above aliased versions $\lambda_i^1, \lambda_i^2, \ldots, \lambda_i^m$ are identified we know that for any substitution σ such that $[\![\sigma(ls_i)]\!] \subseteq [\![x_{s_i}^{-p_i}]\!]$ there exists a substitution $\sigma_{ls_i}^{@\lambda_i^k}$ such that $[\![\sigma(ls_i)]\!] \subseteq [\![\sigma_{ls_i}^{@\lambda_i^k}(ls_i)]\!]$. Thus, in order to check profile satisfaction for the corresponding rewrite rule, we can verify that $\{\!\{\sigma^{@\lambda_i^k}_{ls_i}(rs)\}\!\} \cap [\![p]\!] = \emptyset$ for all $k \in [1, m]$.

Example 4.1. *We consider the signature and the patterns p_1, p_2 in Example 3.1 together with the CBTRS $\mathcal{R}'$ presented in the introduction. In order to check that the rule $concat(cons(e,l),l') \rightarrow cons(e, concat(l,l'))$ satisfies the profile $p_1 * p_1 \mapsto p_1$ we should characterize the substitutions σ such that $\sigma(cons(e,l))$ and $\sigma(l')$ are both p_1-free.*

For each such substitution we should thus have $[\![\sigma(cons(e,l))]\!] \subseteq [\![cons(e,l) \times x^{-p_1}_{List}]\!]$. One can check that we have $[\![cons(e,l) \times x^{-p_1}_{List}]\!] = [\![cons(e\ @\ (x^{-p_1}_{Expr} \setminus lst(l_1)), l\ @\ y^{-p_1}_{List})]\!]$ (we present in the next subsection a method to compute the latter pattern) and we consider the aliased version of $cons(e,l)$, $cons(e\ @\ (x^{-p_1}_{Expr} \setminus lst(l_1)), l\ @\ y^{-p_1}_{List})$. According to Lemma 4.1, σ should be such that $[\![\sigma(e)]\!] \subseteq [\![x^{-p_1}_{Expr} \setminus lst(l_1)]\!]$ and $[\![\sigma(l)]\!] \subseteq [\![y^{-p_1}_{List}]\!]$. Similarly, we have $[\![l' \times z^{-p_1}_{List}]\!] = [\![l'\ @\ z^{-p_1}_{List}]\!]$. Therefore, to satisfy this profile we should show that for all σ such that $[\![\sigma(e)]\!] \subseteq [\![x^{-p_1}_{Expr} \setminus lst(l_1)]\!]$, $[\![\sigma(l)]\!] \subseteq [\![y^{-p_1}_{List}]\!]$ and $[\![\sigma(l')]\!] \subseteq [\![z^{-p_1}_{List}]\!]$, $\sigma(cons(e, concat(l,l')))$ is p_1-free, i.e. that $\{\!\{cons(e@(x^{-p_1}_{Expr} \setminus lst(l_1)), concat(l\ @\ y^{-p_1}_{List}, l'\ @\ z^{-p_1}_{List}))\}\!\} \cap [\![p_1]\!] = \emptyset$.

When verifying the profile $p_2 * p_2 \mapsto p_2$, we can easily check that $[\![cons(e,l) \times x^{-p_2}_{List}]\!] = \emptyset$, i.e. that a list cannot be p_2-free and match $cons(e,l)$, and since there is no substitution which can lead to a p_2-free instance of the pattern in the left-hand side, the profile is satisfied.

We introduce the rewrite system $\mathfrak{R}$ in Figure 1 in order to reduce pattern conjunctions and, in particular, to reduce patterns of the form $q \times x^{-p}_s$, with p a linear regular additive pattern and q a constructor pattern, to either $\perp$ or a sum of aliased versions of q. The rules generally correspond to their counterparts from set theory where constructor patterns correspond to cartesian products and the other extended patterns to the obvious set operations.

The rules A1 and A2, resp. E2 and E3, describe the behaviour of the disjunction, resp. the conjunction, *w.r.t.* $\perp$. Rule E1 indicates that the semantics of a pattern containing a subterm with an empty ground semantics is itself empty. Rule S1 corresponds to the distributivity of cartesian products over disjunction, while rules S2 and S3 express the distributivity of conjunction over disjunction. Finally, rule S4 corresponds to the associativity of the disjunction.

The semantics of a variable of a given sort is the set of all ground constructor patterns of the respective sort. Thus, the difference between the ground semantics of any pattern and the ground semantics of a variable of the same sort is the empty set (rule M1). The rules M2–M6 correspond to set theory laws for complements. M7 corresponds to the set difference of cartesian products; the case when the head symbol is a constant c corresponds to the rule $c \setminus c \Rightarrow \perp$. M8 corresponds to the special case where complemented sets are disjoint.

Rules T1 and T2 just indicate that the intersection with the set of all values (of a sort) as well as the intersection between two identical sets have no effect; we nevertheless use the alias to keep track of the original variable and identify the pattern-freeness constraints on the respective variable (see Proposition 4.3). Rule T3 corresponds to distribution laws for the joint intersection, while T4 corresponds to the disjointed case.

The rules L1 and L2 specify respectively that aliasing a $\perp$ is useless and that aliasing a sum comes to aliasing all its patterns. Rule L3 indicates that taking the complement of an alias variable is nothing else than performing the operation on the aliased term and changing thus accordingly the constraints on the respective variable. Similarly, rules L4 and L5 propagate the alias in the case of conjunctions.

The rules P1 and P2 follow from the observation that the ground semantics of an annotated variable can be also defined as $[\![x^{-p}_s]\!] = \bigcup_{\alpha \in \mathcal{C}_s} [\![\alpha(x^{-p}_{s_1}, \ldots, x^{-p}_{s_i}) \setminus p]\!]$ and that the only relevant element in this set for the conjunction between x^{-p}_s and a pattern $c(v_1, \ldots, v_n)$ is $c(x^{-p}_{s_1}, \ldots, x^{-p}_{s_i})$ (see rules T3 and T4). Moreover, for any symbolic patterns q_1, q_2 and regular additive pattern p we have $[\![(q_1 \setminus p) \times q_2]\!] = [\![(q_1 \times q_2) \setminus p]\!]$ leading to the right-hand side of the rule. Note that ${z_i}^{-p}_{s_i}$ are fresh variables generated automatically and that we keep track of the original variables as in rule T1.

Rules P3, P4 and P5 correspond to the obvious set operation laws but restrict to the only possible cases in the reduction. Finally, we can observe that, thanks to the algorithm introduced in [9], we can determine if $\{\!\{\overline{x}^{-p}_s \setminus \overline{q}\}\!\} = \emptyset$ for any regular additive patterns p and q. Therefore, the system is finalized by the rule P7 which eliminates, when possible, annotated variables. Note that, in order to apply P7 exhaustively, $\mathfrak{R}$ also needs a rule to perform some $\setminus$-factorization around variables, resulting in the rule P6.

Note that Figure 1 defines rule schemas and that the actual rewrite system depends on the signature of the analysed CBTRS: the rule schemas involving symbols expand to the rules obtained by instantiating α, β, δ with all the symbols in $\mathcal{C}$ and T2 expands to two rules corresponding to the two cases. The conditions in rules P3–P7 are evaluated independently of this rewrite system using the method in [9].

It is relatively easy to see that if $p \Longrightarrow^*_{\mathfrak{R}} q$ and p is linear, resp. regular, resp. quasi-additive then, q is linear (all variable duplications occur below different elements of a sum), resp. regular (all new annotated variables are introduced because of a pre-existing annotated variable), resp. quasi-additive (no products introduced and all generated complements have right-hand sides already present in the initial term). All the conjunctions $p \times q$, resp. complements $p \setminus q$, that we have to reduce for the scope of this paper are such that q is linear, and this property is also preserved by $\mathfrak{R}$. Finally, $\mathfrak{R}$ also preserves the ground semantics:

Proposition 4.2 (Semantics preservation). *For any extended patterns p, q, if $p \Longrightarrow^*_{\mathfrak{R}} q$ then $[\![p]\!] = [\![q]\!]$.*

$\mathfrak{R}$ is confluent and terminating, and the normal forms allow the characterization of the substitutions in the inference process:

Proposition 4.3. *The rewrite system $\mathfrak{R}$ is confluent and terminating. Given a pattern $t \in \mathcal{T}_s(\mathcal{C}, \mathcal{X})$ and a linear regular additive pattern p, $v := (t \times x^{-p}_s) \downarrow_{\mathfrak{R}}$ is either $\perp$ or a term of the form $\tau^1 + \tau^2 + \cdots + \tau^m$ with $\tau^i, i \in [1,m]$ aliased versions of t. Moreover, if t is linear, then we have:*

- *$v = \perp \iff [\![t \times x^{-p}_s]\!] = \emptyset$, or*
- *for all σ s.t. $\sigma(t) \in \mathcal{T}(\mathcal{F})$, $[\![\sigma(t)]\!] \subseteq [\![x^{-p}_s]\!] \iff \exists k \in [1,m], [\![\sigma(t)]\!] \subseteq [\![\tau^k]\!]$.*

Remove empty sets:				
(A1)	$\perp + \overline{v}$	$\Rightarrow$	$\overline{v}$	
(A2)	$\overline{v} + \perp$	$\Rightarrow$	$\overline{v}$	
(E1)	$\delta(\overline{v_1}, \ldots, \perp, \ldots, \overline{v_n})$	$\Rightarrow$	$\perp$	
(E2)	$\perp \times \overline{v}$	$\Rightarrow$	$\perp$	
(E3)	$\overline{v} \times \perp$	$\Rightarrow$	$\perp$	
Expand sums:				
(S1)	$\delta(\overline{v_1}, \ldots, \overline{v_i} + \overline{w_i}, \ldots, \overline{v_n})$	$\Rightarrow$	$\delta(\overline{v_1}, \ldots, \overline{v_i}, \ldots, \overline{v_n}) + \delta(\overline{v_1}, \ldots, \overline{w_i}, \ldots, \overline{v_n})$	
(S2)	$(\overline{v_1} + \overline{v_2}) \times \overline{w}$	$\Rightarrow$	$(\overline{v_1} \times \overline{w}) + (\overline{v_2} \times \overline{w})$	
(S3)	$\overline{v} \times (\overline{w_1} + \overline{w_2})$	$\Rightarrow$	$(\overline{v} \times \overline{w_1}) + (\overline{v} \times \overline{w_2})$	
(S4)	$\overline{u} + (\overline{v} + \overline{w})$	$\Rightarrow$	$(\overline{u} + \overline{v}) + \overline{w}$	
Simplify complements:				
(M1)	$\overline{v} \setminus \overline{x}_s^{\overline{\perp}}$	$\Rightarrow$	$\perp$	
(M2)	$\overline{v} \setminus \perp$	$\Rightarrow$	$\overline{v}$	
(M3)	$(\overline{v_1} + \overline{v_2}) \setminus \overline{w}$	$\Rightarrow$	$(\overline{v_1} \setminus \overline{w}) + (\overline{v_2} \setminus \overline{w})$	
(M5)	$\perp \setminus \overline{v}$	$\Rightarrow$	$\perp$	
(M6)	$\alpha(\overline{v_1}, \ldots, \overline{v_n}) \setminus (\overline{v} + \overline{w})$	$\Rightarrow$	$(\alpha(\overline{v_1}, \ldots, \overline{v_n}) \setminus \overline{v}) \setminus \overline{w}$	
(M7)	$\alpha(\overline{v_1}, \ldots, \overline{v_n}) \setminus \alpha(\overline{t_1}, \ldots, \overline{t_n})$	$\Rightarrow$	$\alpha(\overline{v_1} \setminus \overline{t_1}, \ldots, \overline{v_n}) + \cdots + \alpha(\overline{v_1}, \ldots, \overline{v_n} \setminus \overline{t_n})$	
(M8)	$\alpha(\overline{v_1}, \ldots, \overline{v_n}) \setminus \beta(\overline{w_1}, \ldots, \overline{w_m})$	$\Rightarrow$	$\alpha(\overline{v_1}, \ldots, \overline{v_n})$	*with* $\alpha \neq \beta$
Simplify conjunctions:				
(T1)	$\overline{x}_s^{\overline{\perp}} \times \overline{y}_s^{\overline{q}}$	$\Rightarrow$	$\overline{x} \mathbin{@} \overline{y}_s^{\overline{q}}$	
(T2)	$\overline{x}_s^{\overline{p}} \times \overline{y}_s^{\overline{q}}$	$\Rightarrow$	$\overline{x} \mathbin{@} \overline{y}_s^{\overline{p}}$	*with* $\overline{p} = \overline{q} \vee \overline{q} = \perp$
(T3)	$\alpha(\overline{v_1}, \ldots, \overline{v_n}) \times \alpha(\overline{w_1}, \ldots, \overline{w_n})$	$\Rightarrow$	$\alpha(\overline{v_1} \times \overline{w_1}, \ldots, \overline{v_n} \times \overline{w_n})$	
(T4)	$\alpha(\overline{v_1}, \ldots, \overline{v_n}) \times \beta(\overline{w_1}, \ldots, \overline{w_m})$	$\Rightarrow$	$\perp$	*with* $\alpha \neq \beta$
Simplify aliases:				
(L1)	$\overline{x} \mathbin{@} \perp$	$\Rightarrow$	$\perp$	
(L2)	$\overline{x} \mathbin{@} (\overline{v} + \overline{w})$	$\Rightarrow$	$\overline{x} \mathbin{@} \overline{v} + \overline{x} \mathbin{@} \overline{w}$	
(L3)	$(\overline{x} \mathbin{@} \overline{v}) \setminus \overline{w}$	$\Rightarrow$	$\overline{x} \mathbin{@} (\overline{v} \setminus \overline{w})$	
(L4)	$(\overline{x} \mathbin{@} \overline{v}) \times \overline{w}$	$\Rightarrow$	$\overline{x} \mathbin{@} (\overline{v} \times \overline{w})$	
(L5)	$\overline{v} \times (\overline{x} \mathbin{@} \overline{w})$	$\Rightarrow$	$\overline{x} \mathbin{@} (\overline{v} \times \overline{w})$	*with* $\overline{v} \neq \overline{y} \mathbin{@} \overline{u}$
Simplify p-free:				
(P1)	$\overline{x}_s^{\overline{p}} \times \alpha(\overline{v_1}, \ldots, \overline{v_n})$	$\Rightarrow$	$\overline{x} \mathbin{@} (\alpha(z_{1s_1}^{\overline{p}} \times \overline{v_1}, \ldots, z_{ns_n}^{\overline{p}} \times \overline{v_n}) \setminus \overline{p})$	
(P2)	$\alpha(\overline{v_1}, \ldots, \overline{v_n}) \times \overline{x}_s^{\overline{p}}$	$\Rightarrow$	$\alpha(\overline{v_1} \times z_{1s_1}^{\overline{p}}, \ldots, \overline{v_n} \times z_{ns_n}^{\overline{p}}) \setminus \overline{p}$	
(P3)	$\alpha(\overline{v_1}, \ldots, \overline{v_n}) \times (\overline{x}_s^{\overline{p}} \setminus \overline{t})$	$\Rightarrow$	$(\alpha(\overline{v_1}, \ldots, \overline{v_n}) \times \overline{x}_s^{\overline{p}}) \setminus \overline{t}$	if $[\![\overline{x}_s^{\overline{p}} \setminus \overline{t}]\!] \neq \emptyset$
(P4)	$\overline{y}_s^{\overline{q}} \times (\overline{x}_s^{\overline{p}} \setminus \overline{t})$	$\Rightarrow$	$(\overline{y}_s^{\overline{q}} \times \overline{x}_s^{\overline{p}}) \setminus \overline{t}$	if $[\![\overline{x}_s^{\overline{p}} \setminus \overline{t}]\!] \neq \emptyset$
(P5)	$(\overline{x}_s^{\overline{p}} \setminus \overline{t}) \times \overline{v}$	$\Rightarrow$	$(\overline{x}_s^{\overline{p}} \times \overline{v}) \setminus \overline{t}$	if $[\![\overline{x}_s^{\overline{p}} \setminus \overline{t}]\!] \neq \emptyset$
(P6)	$(\overline{x}_s^{\overline{p}} \setminus \overline{t}) \setminus \overline{q}$	$\Rightarrow$	$\overline{x}_s^{\overline{p}} \setminus (\overline{t} + \overline{q})$	if $[\![\overline{x}_s^{\overline{p}} \setminus \overline{t}]\!] \neq \emptyset$
(P7)	$\overline{x}_s^{\overline{p}} \setminus \overline{t}$	$\Rightarrow$	$\perp$	if $[\![\overline{x}_s^{\overline{p}} \setminus \overline{t}]\!] = \emptyset$

Figure 1: $\mathfrak{R}$: reduce patterns of the form $p \times q$; $\overline{u}, \overline{v}, \overline{v_1}, \ldots, \overline{v_n}$, $\overline{w}, \overline{w_1}, \ldots, \overline{w_n}$ range over quasi-additive patterns, $\overline{p}, \overline{q}, \overline{t}$ range over regular additive patterns, $\overline{t_1}, \ldots, \overline{t_n}$ range over symbolic patterns, $\overline{x}, \overline{y}$ range over pattern variables. α, β expand to all the symbols in $\mathcal{C}$, and δ in $\mathcal{C}^{n>0}$.

Thus, given a rule $f^{!\mathcal{P}}(\mathit{ls}_1, \ldots, \mathit{ls}_n) \twoheadrightarrow r$ and a profile $p_1 * \cdots * p_n \mapsto p \in \mathcal{P}$, we can use the $\mathfrak{R}$ to reduce $(\!|\mathit{ls}_1 \times z_1{}^{-p_1}, \ldots, \mathit{ls}_n \times z_n{}^{-p_n}|\!)$ into a sum of tuples of aliased versions of $\mathit{ls}_1, \ldots, \mathit{ls}_n$ (or into $\bot$) and, consequently, to characterize the substitutions σ such that the elements of $\sigma((\!|\mathit{ls}_1, \ldots, \mathit{ls}_n|\!))$ are pairwise p_i-free.

4.3 Establishing pattern-free properties

The substitutions obtained by inference, with the aliasing approach in the previous section, map variables to quasi-symbolic patterns such that, when applied to the right-hand side rs of a rewrite rule, the resulting term is an aliased version of rs. We extend the notion of ground semantics (Definition 3.2) to such aliased versions of terms:

Definition 4.2 (Extended ground semantics). *Given τ, an aliased version of $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,*

- $\llbracket \tau \rrbracket = \bigcup_\omega \bigcup_v \llbracket \tau\, [v]_\omega \rrbracket$ *with* $\omega \in \mathcal{P}os(\tau)$ *such that* $\tau_{|\omega} = f_s^{!\mathcal{P}}(\tau_1, \ldots, \tau_n), f_s^{!\mathcal{P}} \in \mathcal{D}_s$ *and* $v \in \mathcal{T}_s(\mathcal{C})$ *q-free with* $q = \sum_{q' \in \mathcal{Q}} q'$, $\mathcal{Q} = \{p \mid \exists p_1 * \cdots * p_n \mapsto p \in \mathcal{P}$ *s.t.* $\forall i \in [1, n]$, $\{\!\!\{\tau_i\}\!\!\} \cap \llbracket p_i \rrbracket = \emptyset\}$.

Profile satisfaction and thus, semantics preservation, can be verified by computing the intersection between the deep semantics of the instantiated right-hand sides and the ground semantics of the corresponding patterns in the (co-domain of the) profiles:

Proposition 4.4. *Given a linear rule $f^{!\mathcal{P}}(\mathit{ls}_1, \ldots, \mathit{ls}_n) \twoheadrightarrow \mathit{rs}$, a profile $\pi = p_1 * \cdots * p_n \mapsto p \in \mathcal{P}$, and $v := (\!|\mathit{ls}_1 \times z_1{}_{s_1}^{-p_1}, \ldots, \mathit{ls}_n \times z_n{}_{s_n}^{-p_n}|\!) \downarrow_{\mathfrak{R}}$, then:*

- *if $v = \bot$ then the rule satisfies the profile π;*
- *if $v = (\!|\lambda_1^1, \ldots, \lambda_n^1|\!) + (\!|\lambda_1^2, \ldots, \lambda_n^2|\!) + \cdots + (\!|\lambda_1^m, \ldots, \lambda_n^m|\!)$ with $\lambda_i^j, j \in [1, m]$ aliased versions of $\mathit{ls}_i, i \in [1, n]$ then, the rule satisfies π iff $\forall k, \{\!\!\{\sigma_{(\!|\mathit{ls}_1, \ldots, \mathit{ls}_n|\!)}^{@(\!|\lambda_1^k, \ldots, \lambda_n^k|\!)}(\mathit{rs})\}\!\!\} \cap \llbracket p \rrbracket = \emptyset$.*

Example 4.2. *We consider the signature and patterns from Example 3.1 and take $\mathcal{P}_1 = \{\bot \mapsto p_1\}$ and $\mathcal{P}_2 = \{p_1 * p_1 \mapsto p_1\}$. In order to prove that $\mathcal{R}'$ is semantics preserving, we use $\mathfrak{R}$ as stated in Proposition 4.4 to check that the flatten rules satisfy the profile $\bot \mapsto p_1$, and that the concat rules satisfy the profile $p_1 * p_1 \mapsto p_1$.*

For the rewrite rule $\mathit{flatten}(\mathit{nil}) \twoheadrightarrow \mathit{nil}$, since $(\!|\mathit{nil} \times x_{List}^{-p_1}|\!) \downarrow_{\mathfrak{R}} = (\!|\mathit{nil}|\!)$ the obtained substitution $\sigma_{(\!|\mathit{nil}|\!)}^{@(\!|\mathit{nil}|\!)}$ is the identity. We thus simply have to check that the right-hand sie is p_1-free, i.e. that $\{\!\!\{\mathit{nil}\}\!\!\} \cap \llbracket p_1 \rrbracket = \emptyset$, which is obvious. For the rule $\mathit{flatten}(\mathit{cons}(\mathit{int}(n), l)) \twoheadrightarrow \mathit{cons}(\mathit{int}(n), \mathit{flatten}(l))$, given that $(\!|\mathit{cons}(\mathit{int}(n), l) \times x_{List}^{-\bot}|\!) \downarrow_{\mathfrak{R}} = (\!|\mathit{cons}(\mathit{int}(n \mathbin{@} y_{Int}^{-\bot}), l \mathbin{@} z_{List}^{-\bot})|\!) = v_1$, we have $\sigma_{(\!|\mathit{cons}(\mathit{int}(n), l)|\!)}^{@v_1} = \{n \mapsto n \mathbin{@} y_{Int}^{-\bot}, l \mapsto l \mathbin{@} z_{List}^{-\bot}\}$. In order to conclude to the profile satisfaction for this rule, we have to check that $\{\!\!\{\mathit{cons}(\mathit{int}(n \mathbin{@} y_{Int}^{-\bot}), \mathit{flatten}(l \mathbin{@} z_{List}^{-\bot}))\}\!\!\} \cap \llbracket p_1 \rrbracket = \emptyset$. Similarly, for $\mathit{flatten}(\mathit{cons}(\mathit{lst}(l), l')) \twoheadrightarrow \mathit{concat}(\mathit{flatten}(l), \mathit{flatten}(l'))$, we can compute the inferred right-hand side and we should check that $\{\!\!\{\mathit{concat}(\mathit{flatten}(l \mathbin{@} x_{List}^{-\bot}), \mathit{flatten}(l' \mathbin{@} y_{List}^{-\bot}))\}\!\!\} \cap \llbracket p_1 \rrbracket = \emptyset$.

For the rule $\mathit{concat}(\mathit{cons}(e, l), l') \twoheadrightarrow \mathit{cons}(e, \mathit{concat}(l, l'))$, the inferred substitution is slightly more complex than in the previous cases since $(\!|\mathit{cons}(e, l) \times x_1{}_{List}^{-p_1}, l' \times x_2{}_{List}^{-p_1}|\!) \downarrow_{\mathfrak{R}} = (\!|\mathit{cons}(e \mathbin{@} (x_{Expr}^{-p_1} \setminus \mathit{lst}(l_1)), l \mathbin{@} y_{List}^{-p_1}), l' \mathbin{@} z_{List}^{-p_1}|\!)$, thus, in this case, we should check that $\{\!\!\{\mathit{cons}(e \mathbin{@} (x_{Expr}^{-p_1} \setminus \mathit{lst}(l_1)), \mathit{concat}(l \mathbin{@} y_{List}^{-p_1}, l' \mathbin{@} z_{List}^{-p_1}))\}\!\!\} \cap \llbracket p_1 \rrbracket = \emptyset$. For the rule $\mathit{concat}(\mathit{nil}, l) \twoheadrightarrow l$ we proceed similarly, hence we should verify that $\{\!\!\{l \mathbin{@} x_{List}^{-p_1}\}\!\!\} \cap \llbracket p_1 \rrbracket = \emptyset$, which is obvious.

Some of the intersections in the example are clearly empty while for others the proof is less obvious. To systematically check the emptiness of such intersections we use an approach similar to the one in [9] which consists in finding an extended pattern whose semantics is equivalent to that of the instantiated right-hand side, express then the deep semantics of this pattern as a union of ground semantics and finally, check the intersections with the semantics of the profile pattern by computing the corresponding conjunctions using $\mathfrak{R}$.

Note that the ground semantics of a term $f_s^{!\mathcal{P}}(t_1, \ldots, t_n)$ does not only depend on the head symbol $f_s^{!\mathcal{P}}$ and, more precisely, on (the co-domain of) its annotation, as it was the case for the simple profiles in [9], but also on its subterms. We can nevertheless systematically construct for any term an extended pattern which is semantically equivalent:

Definition 4.3 (Semantics equivalent). *Given τ, an aliased version of a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, its* semantics equivalent $\tilde{\tau} \in \mathcal{P}(\mathcal{C}, \mathcal{X}^a)$ *is a quasi-symbolic pattern defined as follows:*

- *if $\tau = c(\tau_1, \ldots, \tau_n)$ with $c \in \mathcal{C}$, then $\tilde{\tau} := c(\tilde{\tau_1}, \ldots, \tilde{\tau_n})$;*
- *if $\tau = f_s^{!\mathcal{P}}(\tau_1, \ldots, \tau_n)$ with $f \in \mathcal{D}$, then $\tilde{\tau} := y \mathbin{@} z_s^{-\diamond\tau}$;*
- *if $\tau = x \mathbin{@} q$, then $\tilde{\tau} := x \mathbin{@} q$;*

where $y, z_s^{-\diamond\tau}$ are fresh variables and $\diamond f_s^{!\mathcal{P}}(\tau_1, \ldots, \tau_n) := \sum_{q' \in \mathcal{Q}'} q'$ with $\mathcal{Q}' = \{p \mid \exists p_1 * \cdots * p_n \mapsto p \in \mathcal{P}$ s.t. $\forall i \in [1, n], \{\!\!\{\tilde{\tau_i}\}\!\!\} \cap \llbracket p_i \rrbracket = \emptyset\}$.

Hence, semantics equivalents are aliased versions of constructor patterns such that:

Proposition 4.5. *Given τ, an aliased version of a $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\llbracket \tau \rrbracket = \llbracket \tilde{\tau} \rrbracket$.*

The notions of deep and ground semantics for general terms used in this paper generalize those in [9] but when restricting to extended patterns they are indistinguishable. Consequently, we can use the method introduced in [9] to express the deep semantics of any extended pattern as a union of ground semantics of quasi-additive patterns: $\{\!\!\{\tilde{\tau}\}\!\!\} = \llbracket r_1 \rrbracket \cup \cdots \cup \llbracket r_m \rrbracket$. We should point out that this exact method is the only one directly borrowed from [9] and that the only possible approximations are situated in the definition of (extended) ground semantics. We can then use $\mathfrak{R}$ to check the emptiness of the intersection between the deep semantics of a semantics equivalent pattern and a profile pattern:

Proposition 4.6. *Given a linear quasi-additive pattern t and a linear regular additive pattern q, we have $p \times t \Longrightarrow_{\mathfrak{R}}^* \bot$ if and only if $\llbracket p \times q \rrbracket = \emptyset$.*

Once we have computed the semantics equivalent of the inferred right-hand side, the intersection between its deep semantics and the ground semantics of the pattern from the annotation (see Proposition 4.4) can then be computed by decomposing the deep semantics into ground semantics and using $\mathfrak{R}$ to check the emptiness, in order to conclude on the profile satisfaction.

Example 4.3. *We consider the inferred right-hand sides obtained in Example 4.2 and check the corresponding intersections.*

Since $l @ y^{-p_1}_{List}$ and $l' @ z^{-p_1}_{List}$ are their own semantics equivalents, in order to compute the semantics equivalent of $rs := cons(e@(x^{-p_1}_{Expr} \setminus lst(l_1)), concat(l @ y^{-p_1}_{List}, l' @ z^{-p_1}_{List}))$, we need to check that $\{\!\![y^{-p_1}_{List}]\!\!\} \cap [\![p_1]\!] = \emptyset$. Using the method from [9] mentioned above, we obtain $\{\!\![y^{-p_1}_{List}]\!\!\} = [\![y^{-p_1}_{List}]\!] \cup [\![x^{-p_1}_{Expr} \setminus lst(l_1)]\!] \cup [\![z^{-p_1}_{Int}]\!]$. As the last two are not of sort List, and since $(y^{-p_1}_{List} \times p_1) \downarrow_{\Re} = \bot$, the intersection with $[\![p_1]\!]$ is indeed empty. The semantics equivalent of $concat(l@y^{-p_1}_{List}, l'@z^{-p_1}_{List})$ is thus $u@x^{-p_1}_{List}$ and consequently, $\tilde{rs} = cons(e@(x^{-p_1}_{Expr} \setminus lst(l_1)), u@ y^{-p_1}_{List})$.

To verify $\{\!\![rs]\!\!\} \cap [\![p_1]\!] = \{\!\![\tilde{rs}]\!\!\} \cap [\![p_1]\!] = \{\!\![cons(x^{-p_1}_{Expr} \setminus lst(l_1), y^{-p_1}_{List})]\!\!\} \cap [\![p_1]\!] = \emptyset$, we decompose again the deep semantics and have to check that $([\![cons(x^{-p_1}_{Expr} \setminus lst(l_1), y^{-p_1}_{List})]\!] \cup [\![x^{-p_1}_{Expr} \setminus lst(l_1)]\!] \cup [\![y^{-p_1}_{List}]\!] \cup [\![z^{-\bot}_{Int}]\!]) \cap [\![p_1]\!] = \emptyset$. Since $(cons(x^{-p_1}_{Expr} \setminus lst(l_1), y^{-p_1}_{List}) \times p_1) \downarrow_{\Re} = \bot$ and $(y^{-p_1}_{List} \times p_1) \downarrow_{\Re} = \bot$, the intersection is empty. Consequently, the rule $concat(cons(e, l), l') \twoheadrightarrow cons(e, concat(l, l'))$ satisfies the profile of concat, and thus, is semantics preserving.

We proceed similarly to show that all the intersections in Example 4.2 are empty and conclude thus, that the CBTRS $\mathcal{R}'$ is semantics preserving.

Note that the approach does not depend on the termination of the underlying CBTRS since the semantics is preserved even for an infinite reduction. Nevertheless, in practice, it is mostly interesting when the system is at least weakly normalizing.

5 CHECKING PATTERN-FREENESS OF NON-LINEAR TERMS

The semantics preservation verification technique proposed here relies on the ground semantics of terms which represent an over-approximation of their potential normal forms and this could obviously lead to some false negatives. One of the reasons for these potential false negatives comes from the way we handle the non-linear right-hand sides of rules: since the semantics of a term with its variables replaced by fresh (distinct) ones is included in the semantics of the original terms, we can linearize, if necessary, the right-hand sides of the rules of the CBTRS and subsequently check that it is semantics preserving.

Example 5.1. *We consider the signature $\Sigma = (\mathcal{S}, \mathcal{F})$ with $\mathcal{S} = \{s_1, s_2\}$ and $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{C} = \{a : s_2, b : s_2, c : s_2 * s_2 \mapsto s_1, d : s_1 \mapsto s_1\}$ and $\mathcal{D} = \{f^{!\mathcal{P}_f} : s_1 \mapsto s_1, g^{!\mathcal{P}_g} : s_1 \mapsto s_2\}$ and the rewrite system*

$$\left\{\begin{array}{lcl} f(c(x,y)) & \twoheadrightarrow & c(x,x) \\ f(d(z)) & \twoheadrightarrow & c(g(z),g(z)) \\ g(c(x,y)) & \twoheadrightarrow & a \\ g(d(z)) & \twoheadrightarrow & b \end{array}\right.$$

It is clear that the normal forms of this rewrite system are $c(a,b)$-free but the method described so far fails in checking that the profile $\mathcal{P}_f = \{\bot \mapsto c(a,b)\}$ is satisfied with $\mathcal{P}_g = \emptyset$. This is because the two occurrences of $g(z)$ in the term $c(g(z), g(z))$ are considered separately as if the global term were linearized and thus, for the second rule we have $[\![c(g(z), g(z))]\!] = [\![c(x,y)]\!]$ which contains the pattern $c(a,b)$ contradicting the profile.

The approach can be customized to take into account some forms of non-linearity and for this, we first adapt the definitions of pattern-freeness and ground semantics (Definitions 3.1 and 4.2), to consider the correlation between the identical subterms headed by a defined symbol. These modifications concern only the terms involving defined symbols; the definitions are unchanged for constructor patterns (and left thus implicit in the new definitions).

Definition 5.1 (Pattern-free terms and Extended semantics). *Given a regular additive pattern p and a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X}^a)$,*

- *t is p-free iff $\forall \omega \in \mathcal{P}os(t)$ such that $t_{|\omega} = f^{!\mathcal{P}}_s(t_1, \ldots, t_n)$ with $f \in \mathcal{D}_s$ and $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ for all $i \in [1,n]$, the term $t\left[f^{!\mathcal{P}}_s(t_1, \ldots, t_n) \mapsto v\right]$ is p-free for all q-free value $v \in \mathcal{T}_s(\mathcal{C})$ with $q = \sum_{q' \in \mathcal{Q}} q'$, $\mathcal{Q} = \{r \mid \exists r_1 * \ldots * r_n \mapsto r \in \mathcal{P}$ s.t. $\forall i \in [1,n], t_i\ r_i\text{-free}\}$;*
- *$[\![t]\!] = \bigcup_\omega \bigcup_v [\![t\left[t_{|\omega} \mapsto v\right]]\!]$ with $\omega \in \mathcal{P}os(t)$ such that $t_{|\omega} = f^{!\mathcal{P}}_s(t_1, \ldots, t_n)$, $f^{!\mathcal{P}}_s \in \mathcal{D}_s, \forall i \in [1,n], t_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, and $v \in \mathcal{T}_s(\mathcal{C})$ q-free with $q = \sum_{q' \in \mathcal{Q}} q'$, $\mathcal{Q} = \{r \mid \exists r_1 * \ldots * r_n \mapsto r \in \mathcal{P}$ s.t. $\forall i \in [1,n], \{\!\![t_i]\!\!\} \cap [\![r_i]\!] = \emptyset\}$;*

with $t\,[u \mapsto v]$ denoting the term t where all occurrences of u have been replaced by v.

While the definitions in the previous sections did not take into account multiple occurrences of a variable when they were below a defined symbol, the above definition does so when a correlation can be identified, *i.e.* when the subterms headed by a defined symbol are identical.

Note that in Definition 5.1 all identical terms are replaced by the same value and thus, the approach described in this section applies effectively only for confluent CBTRSs, leading potentially to false positives for non-confluent systems. Moreover, for some reasons explained intuitively below, the propositions of this section and their proofs assume a strict reduction strategy.

Example 5.2. *We consider the signature from Example 5.1. Using any of the definitions of ground semantics in this paper, we have $[\![c(x_{s_2}, x_{s_2})]\!] = \{c(a,a), c(b,b)\}$. Moreover, the updated definitions in this section recognize that both instances of the variable z_{s_1} in the term $c(g(z_{s_1}), g(z_{s_1}))$ are correlated and consider that, by confluence, both instances of $g(z_{s_1})$ are eventually reduced to a same term. Hence, when considering the profiles $\mathcal{P}_f = \{\bot \mapsto c(a,b)\}$ and $\mathcal{P}_g = \emptyset$, we have $[\![c(g(z_{s_1}), g(z_{s_1}))]\!] = \bigcup_{v \in [\![x_{s_2}]\!]} [\![c(g(z_{s_1}), g(z_{s_1}))\left[g(z_{s_1}) \mapsto v\right]]\!] = \bigcup_{v \in [\![x_{s_2}]\!]} [\![c(v,v)]\!] = [\![c(x_{s_2}, x_{s_2})]\!]$.*

On the other hand, if the occurrences of the same variable are not in a constructor term and not in identical subterms headed by a defined symbol, the definitions consider them as distinct variables. For example, the semantics of the term $c(g(f(x_{s_1})), g(x_{s_1}))$ considers the two occurrences of the variable x_{s_2} as if they were different since there is no way to correlate them through the respective reductions of the terms headed by f and g in the general case. Similarly, if we consider a defined symbol $h : s_1 * s_2 \mapsto s_1$ and the term $h(c(x_{s_2}, x_{s_2}), x_{s_2})$, while the correlation between the instances of the variable x_{s_2} below the constructor symbol c are taken into account as stated before, there is no correlation with the occurrence of x_{s_2} in the second argument of h.

The strong relationship between pattern-freeness and semantics, given by Propositions 3.1 and 3.2, remains valid for these new definitions. Thus, $c(x_{s_2}, x_{s_2})$ and $c(g(z_{s_1}), g(z_{s_1}))$ in the above example are $c(a, b)$ and $c(b, a)$-free.

Due to the way Definition 5.1 handles the different occurrences of a given variable and, in particular, to the lack of correlation between identical variables below different defined symbols, the semantics is preserved only when the substitutions involved in the reduction are value substitutions, *i.e.* in the context of rewriting with a strict reduction strategy. Moreover, the semantics is not necessarily preserved at each intermediate step but eventually retrieved:

Proposition 5.1. *Given a strictly semantics preserving CBTRS $\mathcal{R}$, we have:*

$$\forall u, v \in \mathcal{T}(\mathcal{F}) \text{ s.t. } u \longrightarrow_{\mathcal{R}} v, \exists v' \text{ s.t. } v \longrightarrow^*_{\mathcal{R}} v' \wedge [\![v']\!] \subseteq [\![u]\!]$$

For example, if we consider the system $\mathcal{R}$ presented in Example 5.1 then $c(g(c(a, b)), g(c(a, b))) \longrightarrow_{\mathcal{R}} c(a, g(c(a, b)))$ and, while the latter is not $c(a, b)$-free, its reduct $c(a, a)$ is.

This property guarantees that the normal forms preserve the semantics and consequently, the pattern-freeness properties of the initial term, only when the CBTRS is confluent; our approach applies thus only for confluent CBTRS. For example, if we add to the rewrite system in Example 5.1 the rule $g(c(a, b)) \twoheadrightarrow b$, rendering it non-confluent, we would obtain $c(a, g(c(a, b))) \longrightarrow_{\mathcal{R}} c(a, b)$ with this latter normal form clearly not $c(a, b)$-free; our method reports nevertheless the non-confluent system as profile compliant.

The decomposition of the deep semantics into ground semantics introduced in [9] also applies to non-linear patterns. Moreover, we can easily adapt the construction of the semantics equivalent introduced in Definition 4.3 by simply using, instead of the arbitrary names for the fresh variables, variable names which keep track of the original term, *i.e.* the semantics equivalent of a term $f(t_1, \ldots, t_n)$ is the variable named $\overline{f(t_1, \ldots, t_n)}$.

The inference of the shape of the variables in the left-hand side of a rule, *w.r.t.* value substitutions, can then be performed as before using $\mathfrak{R}$:

Proposition 5.2. *Given a rule $f^{!\mathcal{P}}(l_1, \ldots, l_n) \twoheadrightarrow r$, a profile $\pi = p_1 * \cdots * p_n \mapsto p \in \mathcal{P}$, and $v := (\!| l_1 \times x_{s_1}^{-p_1}, \ldots, l_n \times x_{s_n}^{-p_n} |\!) \downarrow_{\mathfrak{R}}$, then:*

- *if $v = \bot$ then the rule strictly satisfies the profile π;*
- *if $v = \lambda^1 + \cdots + \lambda^m$, with, for all $j \in [1, m]$, $\lambda^j = (\!| \lambda^j_1, \ldots, \lambda^j_n |\!)$ such that λ^j_i aliased versions of l_i, $i \in [1, n]$, we have:* $\forall j, \{\!\{\sigma^{@\lambda^k}_{(\!| l_1, \ldots, l_n |\!)}(r)\}\!\} \cap [\![p]\!] = \emptyset \Longrightarrow$ *the rule strictly satisfies π.*

Similarly, as before, we can verify the emptiness of conjunctions of patterns using $\mathfrak{R}$, but some other cases might appear when non-linear patters are concerned. In particular, since we deal with non-linear terms, a variable can have several occurrences with different aliased patterns in the aliased version, like in the term $c(x@a, x@b)$, indicating an empty semantics.

Proposition 5.3. *Given ρ an aliased version of a constructor pattern r and a linear regular additive pattern q, with $u := (\rho \times q) \downarrow_{\mathfrak{R}}$, we have $[\![\rho \times q]\!] = \emptyset$ if and only if*

- *$u = \bot$, or*
- *$u = \rho_1 + \cdots + \rho_n$, with $\rho_i, i \in [1, n]$, aliased versions of r s.t. $\exists x \in \mathcal{V}ar(r)$ such that $\rho_{i@x} \Longrightarrow^*_{\mathfrak{R}} \bot$.*

The first item corresponds to the linear case (Proposition 4.3) while the second checks if the (semantics of the) different occurrences of the same variable are consistent (*e.g.* check that $[\![a \times b]\!] = \emptyset$ for $c(x \mathbin{@} a, x \mathbin{@} b)$). In the latter, the products generated by the different annotations of a same variable involve only linear patterns, and can indeed be checked using $\mathfrak{R}$:

Proposition 5.4. *Given the linear quasi-additive patterns p and q such that all variables in p and q have the same pattern annotation, $(p \times q) \downarrow_{\mathfrak{R}} = \bot$ iff $[\![p \times q]\!] = \emptyset$.*

Example 5.3. *To prove that $c(x_{s_2}, x_{s_2})$ is $c(a, b)$-free we first express its deep semantics in terms of ground semantics, $\{\!\{c(x_{s_2}, x_{s_2})\}\!\} = [\![c(x_{s_2}, x_{s_2})]\!] \cup [\![x_{s_2}]\!]$, and then check the emptiness for the intersections $[\![x_{s_2}]\!] \cap [\![c(a, b)]\!]$ and $[\![c(x_{s_2}, x_{s_2})]\!] \cap [\![c(a, b)]\!]$. The former is immediately checked since $c(a, b)$ is not a term of s_2. For the latter, we have $c(x_{s_2}, x_{s_2}) \times c(a, b) \Longrightarrow^*_{\mathfrak{R}} c(x \mathbin{@} a, x \mathbin{@} b)$ and since $a \times b \Longrightarrow^*_{\mathfrak{R}} \bot$ we can conclude that $[\![c(x_{s_2}, x_{s_2})]\!] \cap [\![c(a, b)]\!] = [\![c(x_{s_2}, x_{s_2}) \times c(a, b)]\!] = \emptyset$.*

We can proceed similarly for $c(f(x_{s_2}), f(x_{s_2}))$ since its semantics equivalent is $c(fx \mathbin{@} y_{s_2}^{-\bot}, fx \mathbin{@} z_{s_2}^{-\bot})$, with fx the variable generated when computing the semantics equivalent, and the same reductions as above allow us to conclude that the term is $c(a, b)$-free.

Since the profile satisfaction is only established *w.r.t.* value substitutions, the semantics is preserved only when the underlying reduction strategy for the analysed CBTRS is a strict one. In fact, we can prove that unrestricted reductions could be considered with specific constraints on the profiles (*e.g.* when each symbol has at most one profile) but, for simplicity, we have not presented these constraints. We also conjecture the approach is valid for unrestricted strategies.

6 RELATED WORK

This paper proposes a method to verify the absence of specified patterns in the terms reachable by a rewrite relation. Different related work have proposed several approaches to provide similar guarantees on the shape of corresponding normal forms:

Pattern eliminating transformation The general approach and verification method proposed here are based on the notions of pattern-freeness and semantics introduced in [9] to characterize the result of pass-like program transformations. The approach in [9] used simple annotations indicating just the patterns supposed to be absent from the final result without specifying any pre-conditions on the arguments of the analysed function. Moreover, the results had been certified only for right linear CBTRSs. Here, we use annotations describing a set of pre- and post-conditions, thus allowing for a substantially more precise description of the expected behaviour of the reduction associated to each function symbol. For comparison, among the examples provided in the implementation[1], only flatten1, negativeNF and skolemization can be handled with the method in [9] (as shown in the Figure 2, where examples which could be checked in [9] are marked ✓, while new examples which

[1]the source code and examples can be downloaded from http://github.com/plermusiaux/pfree_check and the online version is available at http://htmlpreview.github.io/?https://github.com/plermusiaux/pfree_check/blob/webnix/out/index.html

could be checked only with the approach described here are marked ✚). Non-linearity is also handled now in the case of a strict reduction strategy.

Type based approaches Multiple approaches propose to use type checking to provide varied guarantees to functional programs. In particular, notions such as constructor subtypes [5] could be used to construct complex type systems with some types describing syntactical properties similar to the ones considered here. Comparing to our approach which simply relies on annotating function symbols, defining such type systems is a substantially more involved process. Some approaches, such as refinement types [12] or Liquid Types [25, 30], also propose to enrich type systems with some form of annotation, and can thus be seen as less intrusive alternatives to provide guarantees on the results of functional programs. Some inference mechanisms trying to deduce some of these annotations have also been proposed [1]. Nevertheless, we claim the approach proposed here relies on simpler forms of annotations that are both easier to express (since not needing any declaration of auxiliary types) and understand (since not relying on complex annotation semantics).

Recursion schemes When dealing with higher order functions, some approaches propose to rely on recursion schemes, a form of higher order grammars that are used as tree generators, to describe computation trees [20]. In such approaches, the verification problems are solved by model checking the recursion schemes generated from the given functional program. Higher order recursion schemes have also been extended to include pattern matching [24] and provide the basis for automatic abstraction refinement. While these approaches provide the undeniable advantage of dealing with higher-order programs without additional external inputs, we claim that annotated CBTRSs are easier to grasp when specifying first-order functional programs, and the use of the annotation system also contributes to a more precise (and less intrusive) way to express and control the considered over-approximation.

XML Transformation Exact type-checking of XML processing programs is another field that primarily deals with a similar verification problem of tree transformations. In this context, multiple approaches[23, 29] proposed to use Tree Transducers as the model of XML programming, and use inverse type inference to type-check XML transformations modeled by such Tree Transducers. Kobayashi et al. also introduced in [21] a class of higher order tree transducers which can be modeled by recursion schemes and thus, provided a sound and complete algorithm to solve such verification problem. Other approaches, such as XDuce[19] and CDuce[7], propose to perform type checking of functional XML programs using type-annotation. These approaches rely on a semantics subtyping[13] method fairly similar to the semantics preservation we consider here, but that does not provide support for non right-linear systems. Overall, while these methods can perform similar verification, they are also confined to the XML framework that can be unnecessarily cumbersome when dealing with functional programming.

Tree automata completion Tree automata is a natural approach to describe syntactic properties of tree-shaped terms. When considered in the context of term rewriting system, completion techniques have been defined to compute the set of terms reachable by the rewrite relation as an extended tree automaton [14]. Moreover, tree automata could be used to provide more precise characterizations of the considered terms (and their normal forms) than the pattern-free properties considered here. This approach could, therefore, be applied to provide similar guarantees as the ones presented in this paper, but is nevertheless, constrained by its conditional termination that restricts both the TRS and the set of equational approximations used [15, 27]. Recently, a counter-example based abstraction refinement procedure was proposed as a way to control the over-approximation [17].

We provide in the following table an execution time comparison on some classical scenarios between our implementation, Timbuk3 [14] and Timbuk4 [17]. Note that we only considered cases where the function profiles could be expressed (and thus verified) by our approach. We are indeed limited by the expressiveness of profiles defined using pattern-free properties (which excludes classical scenarios such as ordered trees or size constraints, that can be handled with tree automata completion approaches). However, we argue that pattern profiles are easy to express and understand by a user, and are powerful enough, particularly in a compilation context (where differences between the intermediate languages can, in our experience, be simply and clearly expressed with anti-patterns). Therefore, while the tree automata completion provides a broader approach, our benchmarks show that Timbuk4 [17] is significantly slower, and that Timbuk3 [14] fails (marked ✗) on a few cases, and while faster, is still slower than our implementation:

	pfree check		timbuk 3.2		timbuk 4	
flatten1	✓	$21\mu s$	✗	∞	✓	$685ms$
flatten2	✚	$31\mu s$	✗	∞	✓	$975ms$
negativeNF	✓	$395\mu s$	✓	$3,2ms$	✓	$104s$
skolem	✓	$45\mu s$	✗	$1,5s$	✓	$1,6s$
delete	✚	$107\mu s$	✓	$2,2ms$	✓	$286ms$
reverse	✚	$152\mu s$	✓	$614ms$	✓	$1,4s$
reverse2	✚	$429\mu s$	✓	$714ms$	✓	$2,3s$
insertSort	✚	$211\mu s$	✓	$65ms$	✓	$731ms$
mergeSort	✚	$872\mu s$	✓	$> 1h$	✓	$1,4s$
multiply0	✚	$11\mu s$	✓	$2,4ms$	✓	$225ms$

Figure 2: Comparison table between our method, Timbuk 3.2 and 4. ✓ indicates the properties can be verified and ✗ denotes failure; ✚ is used to indicate the properties could not be verified with the method in [9] but could be verified with the current approach.

The examples in the table are available on GitHub and represent some classical use-cases: *flatten1* and *flatten2* correspond to the systems presented in the introduction, in *negativeNF* we can check that the transformed formulae are in negation normal form, *skolem* allows us to verify that a formula contains no existential quantifier after its skolemisation, for *delete* and *reverse* we can check the expected result for the respective operation for an ordered list while for *reverse2* we verify the result of performing the reverse twice, *insertSort* and *mergeSort* verify that the result is a sorted list, and *multiply0* verifies that multiplying by 0 (in Peano arithmetic) results in 0. In these examples, the most complex patterns in annotations (which are determinant for the complexity of the method) contain 3 symbols and are of depth of 3, and the patterns of the rules of the

CBTRSs contain at most a dozen symbols and up to a depth of 5. The provided repository also contains a branch running benchmarks with random patterns involving symbols of arity at most 6 and going up to a depth of 6; checking a CBTRS of 25 such rules is done in around 1s.

7 CONCLUSIONS

We have proposed a method to statically analyse constructor term rewriting systems and verify the absence of patterns from the corresponding normal forms. The approach is non-intrusive and avoids the burden of specifying a specific language to characterize the result of the transformation, as the user is simply requested to indicate the pattern profiles for the corresponding functions. Although the method is expected to be used for CBTRSs proved terminating [2, 16, 18], the static analysis can be performed regardless of the termination of the system, and still guarantees (for linear CBTRSs) that all the intermediate terms in the reduction are pattern-free. In the non-linear case, the semantics is not necessarily preserved at each reduction step and thus, may not be ultimately retrieved if the system is not at least weakly normalizing.

The method is based on an over-approximation of the potential normal forms defined by the profile annotations and, as such, can result in false negatives when profiles are not sufficiently precise (as was the case when no profile was specified for the function symbol *concat* for the system in the introduction). The user can nevertheless specify not only the patterns that should be eliminated in the result but also the pre-conditions necessary for each of these pattern-free post-conditions and we argue this allows for a fairly precise specification of the corresponding functions in practice. If the analysed CBTRS features non-linear right-hand sides, these could be linearized before applying the general method but this could result in additional false negatives. When the CBTRS is confluent, as is the case for deterministic functional programs, and if a strict reduction strategy is used, the method handles also some form of non-linear right-hand sides. We conjecture that the profile satisfaction under strict strategy implies the preservation of semantics for the normal forms obtained in the general case, and the proof is under investigation.

The method has been implemented in Haskell and the results in terms of expressiveness and efficiency are very encouraging as shown by the table in the previous section.

We are focusing now on extending the method to handle higher-order functions. While the approach could adapt rather straightforwardly for higher-order functions with a unique annotated profile, the problem becomes more convoluted when considering multiple profiles, as proposed here.

REFERENCES

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. 1994. Soft Typing with Conditional Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 163–173. https://doi.org/10.1145/174675.177847

[2] Thomas Arts and Jürgen Giesl. 2000. Termination of term rewriting using dependency pairs. *Theoretical Computer Science* 236, 1-2 (2000), 133–178. https://doi.org/10.1016/S0304-3975(99)00207-8

[3] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press.

[4] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. 2007. Tom: Piggybacking Rewriting on Java. In *International Conference on Term Rewriting and Applications, RTA 2007 (Lecture Notes in Computer Science, Vol. 4533).* Springer, 36–47. https://doi.org/10.1007/978-3-540-73449-9_5

[5] Gilles Barthe and Maria João Frade. 1999. Constructor Subtyping. In *European Symposium on Programming Languages and Systems, ESOP'99 (Lecture Notes in Computer Science, Vol. 1576).* Springer, 109–127. https://doi.org/10.1007/3-540-49099-X_8

[6] Françoise Bellegarde. 1991. Program Transformation and Rewriting. In *International Conference on Rewriting Techniques and Applications, RTA-91 (Lecture Notes in Computer Science, Vol. 488).* Springer, 226–239. https://doi.org/10.1007/3-540-53904-2_99

[7] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 2003*, Colin Runciman and Olin Shivers (Eds.). ACM, 51–63. https://doi.org/10.1145/944705.944711

[8] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. 2005. Abstract Regular Tree Model Checking. In *International Workshop on Verification of Infinite-State Systems, INFINITY 2005 (Electronic Notes in Theoretical Computer Science, Vol. 149)*, Jirí Srba and Scott A. Smolka (Eds.). Elsevier, 37–48. https://doi.org/10.1016/j.entcs.2005.11.015

[9] Horatiu Cirstea, Pierre Lermusiaux, and Pierre-Etienne Moreau. 2020. Pattern Eliminating Transformations. In *International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2020 (Lecture Notes in Computer Science, Vol. 12561)*, Maribel Fernández (Ed.). Springer, 74–92. https://doi.org/10.1007/978-3-030-68446-4_4

[10] Horatiu Cirstea and Pierre-Etienne Moreau. 2019. Generic Encodings of Constructor Rewriting Systems. In *International Symposium on Principles and Practice of Programming Languages, PPDP 2019.* ACM, 8:1–8:12. https://doi.org/10.1145/3354166.3354173

[11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. 2003. The Maude 2.0 System. In *International Conference on Rewriting Techniques and Applications, RTA 2003 (Lecture Notes in Computer Science, Vol. 2706).* Springer, 76–87. https://doi.org/10.1007/3-540-44881-0_7

[12] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI).* ACM, 268–277. https://doi.org/10.1145/113445.113468

[13] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic Subtyping. In *IEEE Symposium on Logic in Computer Science, (LICS 2002).* IEEE Computer Society, 137–146. https://doi.org/10.1109/LICS.2002.1029823

[14] Thomas Genet. 2014. Towards Static Analysis of Functional Programs Using Tree Automata Completion. In *International Workshop on Rewriting Logic and Its Applications, WRLA 2014 (Lecture Notes in Computer Science, Vol. 8663).* Springer, 147–161. https://doi.org/10.1007/978-3-319-12904-4_8

[15] Thomas Genet. 2016. Termination criteria for tree automata completion. *Journal of Logical and Algebraic Methods in Programming* 85, 1 (2016), 3–33. https://doi.org/10.1016/j.jlamp.2015.05.003

[16] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2006. Mechanizing and Improving Dependency Pairs. *Journal of Automatic Reasoning* 37, 3 (2006), 155–203. https://doi.org/10.1007/s10817-006-9057-7

[17] Timothée Haudebourg, Thomas Genet, and Thomas P. Jensen. 2020. Regular language type inference with term rewriting. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 112:1–112:29. https://doi.org/10.1145/3408994

[18] Nao Hirokawa and Aart Middeldorp. 2005. Automating the dependency pair method. *Information and Computation* 199, 1-2 (2005), 172–199. https://doi.org/10.1016/j.ic.2004.10.004

[19] Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.* 3, 2 (2003), 117–148. https://doi.org/10.1145/767193.767195

[20] Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009.* ACM, 416–428. https://doi.org/10.1145/1480881.1480933

[21] Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. 2010. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010.* ACM, 495–508. https://doi.org/10.1145/1706299.1706355

[22] David Lacey and Oege de Moor. 2001. Imperative Program Transformation by Rewriting. In *International Conference on Compiler Construction, CC 2001 (Lecture Notes in Computer Science, Vol. 2027)*, Reinhard Wilhelm (Ed.). Springer, 52–68. https://doi.org/10.1007/3-540-45306-7_5

[23] Tova Milo, Dan Suciu, and Victor Vianu. 2003. Typechecking for XML transformers. *J. Comput. Syst. Sci.* 66, 1 (2003), 66–97. https://doi.org/10.1016/S0022-0000(02)00030-2

[24] C.-H. Luke Ong and Steven J. Ramsay. 2011. Verifying higher-order functional programs with pattern-matching algebraic data types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011.* ACM, 587–598.

https://doi.org/10.1145/1926385.1926453

[25] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[26] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[27] Toshinori Takai. 2004. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *International Conference on Rewriting Techniques and Applications, RTA 2004 (Lecture Notes in Computer Science, Vol. 3091)*. Springer, 119–133. https://doi.org/10.1007/978-3-540-25979-4_9

[28] Terese. 2003. *Term Rewriting Systems*. Cambridge University Press. M. Bezem, J. W. Klop and R. de Vrijer, eds.

[29] Akihiko Tozawa. 2006. XML Type Checking Using High-Level Tree Transducer. In *International Symposium on Functional and Logic Programming, FLOPS 2006 (Lecture Notes in Computer Science, Vol. 3945)*, Masami Hagiya and Philip Wadler (Eds.). Springer, 81–96. https://doi.org/10.1007/11737414_7

[30] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: experience with refinement types in the real world. In *ACM SIGPLAN symposium on Haskell*, Wouter Swierstra (Ed.). ACM, 39–51. https://doi.org/10.1145/2633357.2633366

[31] Eelco Visser. 1999. Strategic Pattern Matching. In *International Conference on Rewriting Techniques and Applications, RTA-99 (Lecture Notes in Computer Science, Vol. 1631)*. Springer, 30–44. https://doi.org/10.1007/3-540-48685-2_3