Pattern eliminating transformations

Horatiu Cirstea, Pierre Lermusiaux, Pierre-Etienne Moreau

Pattern eliminating transformations

Horatiu Cirstea[1,2], Pierre Lermusiaux[1,2], and Pierre-Etienne Moreau[1,2]

[1] Université de Lorraine
[2] LORIA {name}.{surname}@loria.fr

Abstract. Program transformation is a common practice in computer science, and its many applications can have a range of different objectives. For example, a program written in an original high level language could be either translated into machine code for execution purposes, or towards a language suitable for formal verification. Such compilations are split into several so-called passes which generally aim at eliminating certain constructions of the original language to get a program in some intermediate languages and finally generate the target code. Rewriting is a widely established formalism to describe the mechanism and the logic behind such transformations. In a typed context, the underlying type system can be used to give syntactic guarantees on the shape of the results obtained after each pass, but this approach could lead to an accumulation of auxiliary types that should be considered. We propose in this paper a less intrusive approach based on simply annotating the function symbols with the (anti-)patterns the corresponding transformations are supposed to eliminate. We show how this approach allows one to statically check that the rewrite system implementing the transformation is consistent with the annotations and thus, that it eliminates the respective patterns.

Keywords: Rewriting, Pattern-matching, Pattern semantics, Compilation

1 Introduction

Rewriting is a well established formalism widely used in both computer science and mathematics. It has been used, for example, in semantics in order to describe the meaning of programming languages [27], but also in automated reasoning when describing, by inference rules, a logic, a theorem prover [21], or a constraint solver [20]. Rewriting has turned out to be particularly well adapted to describe program semantics [30] and program transformations [26,7]. There are several languages and tools implementing the notions of pattern matching and rewriting rules ranging from functional languages, featuring relatively simple patterns and fixed rewriting strategies, to rule based languages like Maude [10], Stratego [33], or Tom [5], providing equational matching and flexible strategies; they have been all used as underlying languages for more or less sophisticated compilers.

In the context of compilation, the complete transformation is usually performed in multiple phases, also called passes, in order to eventually obtain a

program in a different target language. Most of these passes concern transformations between some intermediate languages and often aim at eliminating certain constructions of the original language. These transformations could eliminate just some symbols, like in desugaring passes for example, or more elaborate constructions, like in code optimization passes.

To guarantee the correctness of the transformations we could of course use runtime assertions but static guarantees are certainly preferable. When using typed languages, the types guarantee the correctness of some of the constraints on the target language. In this case, the type of the function implicitly expresses the expected result of the transformation. The differences between the source and the target language concern generally only a small percentage of the symbols, and the definition of the target language is often tedious and contains a lot of the symbols from the source type. For example, for a pass performing desugaring we would have to define a target language using the same symbols as the source one but the syntactic sugar symbols.

Formalisms such as the one introduced for NanoPass [22] have proposed a method to eliminate a lot of the overhead induced by the definition of the intermediate languages by specifying only the symbols eliminated from the source language and generating automatically the corresponding intermediate language.

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

$$\begin{array}{l} Expr = int(Int) \\ \quad \mid str(String) \\ \quad \mid lst(List) \end{array} \qquad \begin{array}{l} List = nil \\ \quad \mid cons(Expr, List) \end{array}$$

If, for some reason, we want to define a transformation encoding integers by strings then, the target language in NanoPass would be $Expr^{-int}$, *i.e.* expressions build out of strings and lists. Note that in this case the tool (automatically) removes the symbol *int* from *Expr* and replaces accordingly *Expr* with the new type in the type of *cons*.

This kind of approaches reach their limitations when the transformation of the source language goes beyond the removal of some symbols. For example, if we want to define a transformation which flattens the list expressions and ensures thus that there is no nested list, the following target type should be considered:

$$\begin{array}{l} Expr = lit(Literal) \\ \quad \mid lst(List) \end{array} \qquad \begin{array}{l} Literal = int(Int) \\ \quad \mid str(String) \end{array} \qquad \begin{array}{l} List = nil \\ \quad \mid cons(Literal, List) \end{array}$$

Functional approaches to transformation [29] relying on the use of fine grained type systems which combine overloading, subtyping and polymorphism through the use of variants [13] can be used to define the transformation and perform (implicitly) such verifications. While effective, this method requires to design such adjusted types in a case by case basis.

We propose in this paper a formalism where function symbols are simply annotated with the patterns that should be eliminated by the corresponding transformation and a mechanism to statically verify that the rewriting system implementing the function eliminates indeed these patterns. The method is minimally intrusive: for the above example, we should just annotate the flattening

function symbol with the (anti-)pattern $cons(lst(l_1), l_2)$ and the checker (implemented in Haskell) verifies that the underlying rewriting system is consistent with the annotation, or exhibits the problematic rule(s) and issue(s) if it is not. The method applies to constructor based term rewriting systems which correspond to functional programs where functions are defined by pattern matching, programs which are very common and often used when defining transformations.

First, in the next section, we introduce the basic notions and notations used in the paper. We introduce then, in Section 3, the notion of pattern-free terms together with their ground semantics and we state the pattern-free properties a rewriting system should satisfy to be consistent with the pattern annotations. Section 4 describes a method for automatically checking pattern-free properties relying on the deep semantics, an extension of the ground semantics, and shows how this method can be used to verify that a rewriting system is consistent with the pattern annotations and thus, that specific patterns are absent from the result of the corresponding transformation. We finally present some related work and conclude. All proofs are available in the appendix.

2 Preliminary notions

We define in this section the basic notions and notations used in this paper; more details can be found in [4,32].

A *many-sorted signature* $\Sigma = (\mathcal{S}, \mathcal{F})$, consists of a set of sorts $\mathcal{S}$ and a set of symbols $\mathcal{F}$. The set of symbols is partitioned into two disjoint sets $\mathcal{F} = \mathcal{D} \cup \mathcal{C}$; $\mathcal{D}$ is the set of *defined symbols* and $\mathcal{C}$ the set of *constructors*. A symbol f with *domain* $\mathcal{D}om(f) = s_1 \times \cdots \times s_n \in \mathcal{S}^*$ and *co-domain* $\mathcal{C}o\mathcal{D}om(f) = s \in \mathcal{S}$ is written $f{:}s_1 \times \cdots \times s_n \mapsto s$; 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 $x{:}s$ or x_s to indicate that 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.

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 $f(t_1, \ldots, t_n)$ is in $\mathcal{T}_s(\mathcal{F}, \mathcal{X})$ whenever $f{:}s_1 \times \cdots \times s_n \mapsto s$ and $t_i \in \mathcal{T}_{s_i}(\mathcal{F}, \mathcal{X})$, $i \in [1, n]$. We write $t{:}s$ to 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 terms of sort s and $\mathcal{T}(\mathcal{F})$ denotes the set of all ground terms. Terms in $\mathcal{T}(\mathcal{C})$ are called *values*. A *linear* term is a term where every variable occurs at most once. The linear terms in $\mathcal{T}(\mathcal{C}, \mathcal{X})$ are called *constructor patterns* or simply *patterns*.

A *position* of a term t is a 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. $t_{|\omega}$, resp. $t(\omega)$, denotes the subterm of t, resp. the symbol of t, at position ω. $t\,[s]_\omega$ denotes the term t with the subterm at position ω replaced by s. $\mathcal{P}os(t)$ denotes 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 called its domain. A substitution σ extends as expected to an endomorphism σ' of $\mathcal{T}(\mathcal{F},\mathcal{X})$. To simplify the notations, we do not make the distinction between σ and σ'. Sorted substitutions are such that if $x{:}s$ then $\sigma(x) \in \mathcal{T}_s(\mathcal{F},\mathcal{X})$. Note that for any such sorted substitution σ, $t{:}s$ iff $\sigma(t){:}s$. In what follows we will only consider such sorted substitutions.

Given a sort s, a value $v : s$ and a constructor pattern p, we say that p *matches* v (denoted $p \lessdot v$) if it exists a substitution σ such that $v = \sigma(p)$. Since p is linear, we can give an inductive definition to the pattern matching relation:

$$x \lessdot v \qquad\qquad x \in \mathcal{X}_s$$
$$c(p_1,\ldots,p_n) \lessdot c(v_1,\ldots,v_n) \text{ iff } \wedge_{i=1}^{n}\, p_i \lessdot v_i, \text{ for } c \in \mathcal{C}$$

Starting from the observation that a pattern can be interpreted as the set of its instances, the notion of *ground semantics* was introduced in [9] as the set of all ground constructor instances of a pattern $p \in \mathcal{T}_s(\mathcal{C},\mathcal{X})$: $[\![p]\!] = \{\sigma(p) \mid \sigma(p) \in \mathcal{T}_s(\mathcal{C})\}$. It was shown [9] that, given a pattern p and a value v, $v \in [\![p]\!]$ iff $p \lessdot v$. We denote by $\perp$ the pattern whose semantics is empty, *i.e.* matching no term.

A *constructor rewrite rule* (over Σ) is a pair of terms $\varphi(l_1,\ldots,l_n) \twoheadrightarrow r \in \mathcal{T}_s(\mathcal{F},\mathcal{X}) \times \mathcal{T}_s(\mathcal{F},\mathcal{X})$ with $s \in \mathcal{S}$, $\varphi \in \mathcal{D}$, $l_1,\ldots,l_n \in \mathcal{T}(\mathcal{C},\mathcal{X})$ and such that $\varphi(l_1,\ldots,l_n)$ 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 *rewriting relation* over $\mathcal{T}(\mathcal{F})$, denoted by $\longrightarrow_{\mathcal{R}}$ and such that $t \longrightarrow_{\mathcal{R}} t'$ iff there exist $l \twoheadrightarrow r \in \mathcal{R}$, $\omega \in \mathcal{P}os\,(t)$ and a substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t\,[\sigma(r)]_{\omega}$. The reflexive and transitive closure of $\longrightarrow_{\mathcal{R}}$ is denoted by $\longrightarrow\!\!\!\!\rightarrow_{\mathcal{R}}$.

3 Pattern-free terms and corresponding semantics

We want to ensure that the normal form of a term, if it exists, does not contain a specific constructor and more generally that no subterm of this normal form matches a given pattern. The sort of the term provides some information on the shape of the normal forms since the precise language of the values of a given sort is implicitly given by the signature. Sometimes the normal forms satisfy constraints stronger than those induced from the sorts but these constraints cannot always be determined statically only from the sorts but also depend on the underlying CBTRS.

To guarantee these constraints we annotate all defined symbols with the patterns that are supposed to be absent when reducing a term headed by the respective symbol and we check that the CBTRS defining the corresponding functions are consistent with these annotations.

We focus first on the notion of pattern-free term and on the corresponding ground semantics, and explain in the next sections how one can check pattern-freeness and verify the consistence of the symbol annotations with a CBTRS.

3.1 Pattern-free terms

We consider that every defined symbol $f^{-p} \in \mathcal{D}$ is now annotated with a pattern $p \in \mathcal{T}_{\perp}(\mathcal{C},\mathcal{X}) = \mathcal{T}(\mathcal{C},\mathcal{X}) \cup \{\perp\}$ and we use this notation to define *pattern-free*

terms. Intuitively, any term obtained by reducing a ground term of the form $f^{-p}(t_1, \ldots, t_n)$ contains no subterms matched by p; in particular, if the term is eventually reduced to a value then this value contains no subterms matched by p. Given the example from the introduction, we can consider two function symbols, $flattenE^{-p} : Expr \mapsto Expr$ and $flattenL^{-p} : List \mapsto List$, with $p = cons(lst(l1), l2)$, to indicate that the normal forms of any term headed by one of these symbols contain no nested lists. The annotation of the function symbol for the concatenation, $concat^{-\bot} : List \times List \mapsto List$, indicates that no particular shape is expected for the reducts of the corresponding terms.

Definition 3.1 (Pattern-free terms). *Given p, a constructor pattern or $\bot$,*

- *a value $v \in \mathcal{T}(\mathcal{C})$ is p-free iff $\forall \omega \in \mathcal{P}os\,(v)\,, p \not\trianglelefteq v_{|\omega}$;*
- *a term $u \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ is p-free iff $\forall \sigma$ such that $\sigma(u) \in \mathcal{T}(\mathcal{C})$, $\sigma(u)$ is p-free;*
- *a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is p-free iff $\forall \omega \in \mathcal{P}os\,(t)$ such that $t(\omega) = f_s^{-q} \in \mathcal{D}$, $t\,[v]_\omega$ is p-free for all q-free value $v \in \mathcal{T}_s(\mathcal{C})$.*

A value is p-free if and only if p matches no subterm of the value. For terms containing no defined symbols, 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 replacing (all) the subterms headed by a defined symbol f_s^{-q} by any q-free value of the same sort s results in a p-free term. Intuitively, this corresponds to considering an over-approximation of the set of potential normal forms of an annotated term. While pattern-free properties can be checked for any value by exploring all its subterms, this is not possible for a general term since the property has to be verified by a potentially infinite number of values. We present in Section 4 an approach for solving this problem.

3.2 Generalized ground semantics

The notion of ground semantics presented in Section 2 and, in particular, the approach proposed in [9] to compute differences (and thus intersections) of such semantics, can be used to compare the shape of two constructor patterns p, q (at the root position). More precisely, when $[\![p]\!] \cap [\![q]\!] = \emptyset$ we have that $\forall \sigma, \sigma(q) \notin [\![p]\!]$ and therefore, we can establish that $\forall \sigma, p \not\trianglelefteq \sigma(q)$. We can thus compare the semantics of a given pattern p with the semantics of each of the subterms of a constructor pattern t in order to check that t is p-free.

Example 3.1. Consider the signature Σ with $\mathcal{S} = \{s_1, s_2, s_3\}$ and $\mathcal{F} = \mathcal{C} = \{c_1 : s_2 \times s_1 \mapsto s_1, c_2 : s_3 \mapsto s_1, c_3 : s_1 \mapsto s_2, c_4 : s_3 \mapsto s_2, c_5 : s_3 \mapsto s_3, c_6 :\mapsto s_3\}$. We can compute $[\![c_1(c_4(c_6), y_{s_1})]\!] \cap [\![c_1(x_{s_2}, c_2(c_6))]\!] = [\![c_1(c_4(c_6), c_2(c_6))]\!]$ and thus neither $c_1(c_4(c_6), y_{s_1})$ is $c_1(x, c_2(c_6))$-free nor $c_1(x_{s_2}, c_2(c_6))$ is $c_1(c_4(c_6), y)$-free. Similarly, we can check that $[\![c_3(c_2(z_{s_3}))]\!] \cap [\![c_4(z_{s_3})]\!] = \emptyset$ and that $[\![c_2(z_{s_3})]\!] \cap [\![c_4(z_{s_3})]\!] = \emptyset$ and, as a term of sort s_3 can only contain constructors c_5 and c_6, we can deduce that $c_3(c_2(z_{s_3}))$ is $c_4(z)$-free.

We want to establish a general method to verify pattern-free properties for any term and we propose an approach which relies on the notion of ground semantics extended in order to take into account all terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$:

Definition 3.2 (Generalized ground semantics). *Given a term* $u \in \mathcal{T}(\mathcal{C},\mathcal{X})$, *and a term* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$,

- $[\![u]\!] = \{\sigma(u) \mid \forall \sigma, \sigma(u) \in \mathcal{T}(\mathcal{C})\}$;
- $[\![t]\!] = \{u \in [\![t\,[v]_\omega]\!] \mid \forall \omega \in \mathcal{P}os\,(t) \;\; s.t.\; t(\omega) = f_s^{-p} \in \mathcal{D}, \forall v \in \mathcal{T}_s(\mathcal{C})\; p\text{-free}\}$.

Note that the ground semantics of a variable x_s is the set of all possible ground patterns of the corresponding sort: $[\![x_s]\!] = \mathcal{T}_s(\mathcal{C})$, and for patterns, since they are linear, we can use a recursive definition for the non-variable patterns: $[\![c(p_1,\ldots,p_n)]\!] = \{c(v_1,\ldots,v_n) \mid (v_1,\ldots,v_n) \in [\![p_1]\!] \times \ldots \times [\![p_n]\!]\}$ for all $c \in \mathcal{C}$.

Moreover, by definition we have $[\![f_s^{-p}(t_1,\ldots,t_n)]\!] = \{v \in \mathcal{T}_s(\mathcal{C}) \mid v\ p\text{-free}\}$. The generalized ground semantics of a term rooted by a defined symbol represents an over-approximation of all the possible values obtained by reducing the term with respect to a CBTRS preserving the pattern-free properties.

Pattern-freeness can be checked by exploring the semantics of the term:

Proposition 3.1. *Let* $t \in \mathcal{T}(\mathcal{F},\mathcal{X}), p \in \mathcal{T}_\perp(\mathcal{C},\mathcal{X})$, *t is p-free iff* $\forall v \in [\![t]\!], v$ *is p-free.*

For convenience, we consider also annotated variables whose semantics is that of any term headed by a defined symbol with the same co-domain as the sort of the variable:

$$[\![x_s^{-p}]\!] = \{v \in \mathcal{T}_s(\mathcal{C}) \mid v\ p\text{-free}\}$$

Thus, $[\![f_s^{-p}(t_1,\ldots,t_n)]\!] = [\![x_s^{-p}]\!]$ for all $f_s^{-p} \in \mathcal{D}_s$. Note that $x_s^{-\perp}$ has the same semantics as x_s. We denote by $\mathcal{X}^a$ the set of annotated variables.

Given a linear term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, we can systematically construct its *symbolic equivalent* $\tilde{t} \in \mathcal{T}(\mathcal{C},\mathcal{X}^a)$ by replacing all the subterms of t headed by a defined symbol f_s^{-p} by a fresh variable x_s^{-p} of the corresponding sort and annotated by the same pattern:

Proposition 3.2. $\forall t \in \mathcal{T}(\mathcal{F},\mathcal{X}), [\![t]\!] = [\![\tilde{t}]\!]$

Example 3.2. We consider the signature from Example 3.1 enriched with the defined symbols $\mathcal{D} = \{f^{-p_1} : s_1 \mapsto s_1, g^{-p_2} : s_2 \mapsto s_2\}$ with $p_1 = c_1(c_4(z), y)$ and $p_2 = c_4(z)$. If we consider the term $r_1 = c_1(g^{-p_2}(x), f^{-p_1}(y))$, to construct its symbolic equivalent, we replace $f^{-p_1}(y)$ and $g^{-p_2}(x)$ by $y_{s_1}^{-p_1}$ and $x_{s_2}^{-p_2}$, respectively. Thus we have $\tilde{r_1} = c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})$.

We can thus restrict in what follows to patterns using annotated variables and we consider *extended patterns* built out of this kind of patterns:

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

with $x \in \mathcal{X}_s^a, p, p_1, p_2 : s$ for some $s \in \mathcal{S}, c : s_1 \times \cdots \times s_n \mapsto s \in \mathcal{C}$ and $\forall i \in [1,n], q_i : s_i$.

The pattern matching relation can be extended to take into account disjunctions, conjunctions and complements of patterns:

$$p_1 + p_2 \npreccurlyeq v \text{ iff } p_1 \npreccurlyeq v \;\vee\; p_2 \npreccurlyeq v \qquad p_1 \times p_2 \npreccurlyeq v \text{ iff } p_1 \npreccurlyeq v \;\wedge\; p_2 \npreccurlyeq v$$
$$p_1 \setminus p_2 \npreccurlyeq v \text{ iff } p_1 \npreccurlyeq v \;\wedge\; p_2 \not\npreccurlyeq v \qquad \perp \not\npreccurlyeq v$$

Intuitively, 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. $\bot$ matches no term. $\times$ has a higher priority than $\setminus$ which has a higher priority than $+$.

Extended patterns can share variables but not below a constructor symbol. This corresponds to the fact that p_1 and p_2, in $p_1 + p_2$ (resp. $p_1 \setminus p_2, p_1 \times p_2$), represent independent alternatives *w.r.t.* matching and thus, that their variables are unrelated. For example, the patterns $c_3(c_2(x)) + c_4(x)$ and $c_3(c_2(x)) + c_4(y)$ both represent all values rooted by c_3 followed by c_2, or rooted by c_4.

The notion of ground semantics extends to such patterns by considering the above recursive definition for patterns headed by constructor symbols and

$$[\![p_1 + p_2]\!] = [\![p_1]\!] \cup [\![p_2]\!] \qquad [\![p_1 \setminus p_2]\!] = [\![p_1]\!] \setminus [\![p_2]\!]$$
$$[\![p_1 \times p_2]\!] = [\![p_1]\!] \cap [\![p_2]\!] \qquad [\![\bot]\!] = \emptyset$$

We still have that given an extended pattern p and a value v, $v \in [\![p]\!]$ iff $p \lessdot v$ [9].

In this context, if an extended pattern contains no $\bot$ it is called *pure*, if it contains no $\times$ and no $\setminus$ it is called *additive*, and if it contains no $+$, no $\times$ and no $\setminus$, *i.e.* a term of $\mathcal{T}(\mathcal{C}, \mathcal{X}^a)$, it is called *symbolic*. We call *regular* patterns that contain only variables of the form $x^{-\bot}$. And finally, 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.

We can remark that p_1 and p_2 in Example 3.2 are regular patterns, that $x_{s_2}^{-p_1} \setminus p_2$ is a quasi-additive pattern, and that $\tilde{r_1}$ is a symbolic pattern (indeed, the symbolic equivalent of any term is a symbolic pattern).

3.3 Semantics preserving CBTRS

Generalized ground semantics rely on the symbol annotations and assume thus a specific shape for the 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. *A rewrite rule $l \twoheadrightarrow r$ is* semantics preserving *iff $[\![r]\!] \subseteq [\![l]\!]$. A CBTRS is* semantics preserving *iff all its rewrite rules are.*

Semantics preservation carries over to the induced rewriting relation:

Proposition 3.3. *Given a semantics preserving CBTRS $\mathcal{R}$ we have*
$$\forall t, v \in \mathcal{T}(\mathcal{F}),\ \textit{if } t \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}} v,\ \textit{then } [\![v]\!] \subseteq [\![t]\!].$$

As an immediate consequence we obtain the pattern-free preservation:

Corollary 3.1. *Given a semantics preserving CBTRS $\mathcal{R}$ we have*
$$\forall t, v \in \mathcal{T}(\mathcal{F}), p \in \mathcal{T}(\mathcal{C}, \mathcal{X}),\ \textit{if } t \textit{ is } p\textit{-free and } t \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}} v,\ \textit{then } v \textit{ is } p\textit{-free.}$$

Note that the rules of a CBTRS are of the form $f^{-p}(l_1, \ldots, l_n) \twoheadrightarrow r$ and thus, as an immediate consequence of Definition 3.2, the semantics of the left-hand side of the rewrite rule is the set of all p-free values. Therefore, according to Proposition 3.1, such a rule is semantics preserving if and only if its right-hand side r is p-free. We will see in the next section how pattern-freeness and thus, semantics preservation, can be statically checked.

Example 3.3. We consider the signature from Example 3.2 and the CBTRS:

$$f^{-p_1}(c_1(x,y)) \twoheadrightarrow c_1(g^{-p_2}(x), f^{-p_1}(y)) \qquad g^{-p_2}(c_4(z)) \twoheadrightarrow c_3(c_2(z))$$
$$f^{-p_1}(c_2(z)) \twoheadrightarrow c_2(z) \qquad g^{-p_2}(c_3(y)) \twoheadrightarrow c_3(f^{-p_1}(y))$$

We have seen in Example 3.1 that $c_3(c_2(x))$ is p_2-free and we can thus conclude that the rule $g(c_4(z)) \twoheadrightarrow c_3(c_2(z))$ is semantics preserving. In order to verify in a systematic way the corresponding pattern-free properties of all right-hand sides and conclude that the CBTRS is semantics preserving, we introduce in the next section a method to statically check pattern-freeness.

4 Deep semantics for pattern-free properties

The ground semantics was used in [9] as a means to represent a potentially infinite number of instances of a term in a finite manner and can be employed to check that a pattern matches (or not) a term by computing the intersection between their semantics. For pattern-freeness, we should check not only that the term is not matched by the pattern but also that none of its subterms is matched by this pattern. We would thus need a notion of ground semantics closed by the subterm relation.

We introduce next an extended notion of ground semantics satisfying the above requirements, show how it can be expressed in terms of ground semantics, and provide a method for checking the emptiness of the intersection of such semantics and thus, assert pattern-free properties.

4.1 Deep semantics

The notion of *deep semantics* is introduced to provide more comprehensive information on the shape of the (sub)terms compared to the ground semantics which describes essentially the shape of the term at the root position.

Definition 4.1 (Deep semantics). *Let t be an extended pattern, its deep semantics $\{\!|t|\!\}$ is defined as follows:*

$$\{\!|t|\!\} = \{u_{|\omega} \mid u \in [\![t]\!], \omega \in \mathcal{P}os(u)\}$$

Note first that, similarly to the case of generalized ground semantics, it is obvious that we can always exhibit a symbolic pattern equivalent in terms of deep semantics to a given term, *i.e.* $\forall t \in \mathcal{T}(\mathcal{F},\mathcal{X}), \{\!|t|\!\} = \{\!|\tilde{t}|\!\}$; consequently, we can focus on the computation of the deep semantics of extended patterns. Following this observation and as an immediate consequence of the definition we have a necessary and sufficient condition with regards to pattern-free properties:

Proposition 4.1 (Pattern-free vs Deep Semantics). *Let $p \in \mathcal{T}(\mathcal{C},\mathcal{X}), t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, t is p-free iff $\{\!|\tilde{t}|\!\} \cap [\![p]\!] = \emptyset$.*

To check the emptiness of the above intersection we express the deep semantics of a term as a union of ground semantics and then check for each of them that the intersection with the semantics of the considered pattern is empty.

First, since the deep semantics is based on the generalized ground semantics, we can easily establish a similar recursive definition for constructor patterns:

Proposition 4.2. *For any constructor symbol $c \in \mathcal{C}$ and extended patterns $t_1, \ldots, t_n$, such that $\mathcal{D}om\,(c) = s_1 \times \cdots \times s_n$ and $t_1 : s_1, \ldots, t_n : s_n$, we have:*

- *If $\forall i \in [1,n], [\![t_i]\!] \neq \emptyset$, then $\{\!\{c(t_1, \ldots, t_n)\}\!\} = [\![c(t_1, \ldots, t_n)]\!] \cup \Big(\bigcup_{i=1}^{n} \{\!\{t_i\}\!\}\Big)$;*
- *If $\exists\, i \in [1,n], [\![t_i]\!] = \emptyset$, then $\{\!\{c(t_1, \ldots, t_n)\}\!\} = \emptyset$.*

If we apply the above equation for the non-empty case recursively we eventually have to compute the deep semantics of annotated variables. For this, we use the algorithm introduced in Figure 1: given an annotated variable x_s^{-p}, $\mathsf{getReachable}(s, p, \emptyset, \bot)$ computes a set of pairs $\{(s'_1, p'_1), \ldots, (s'_n, p'_n)\}$ such that $\{\!\{x_s^{-p}\}\!\} = [\![x_{s'_1}^{-p} \setminus p'_1]\!] \cup \ldots \cup [\![x_{s'_n}^{-p} \setminus p'_n]\!]$.

Intuitively, the algorithm uses the definition of the deep semantics of a variable $\{\!\{x_s^{-p}\}\!\} = \{u_{|\omega} \mid u \in [\![x_s^{-p}]\!], \omega \in \mathcal{P}os\,(u)\}$ and the observation that the ground semantics of an annotated variable can be also defined as:

$$[\![x_s^{-p}]\!] = \bigcup_{c \in \mathcal{C}_s} [\![c(x_{s_1}^{-p}, \ldots, x_{s_i}^{-p}) \setminus p]\!] \tag{1}$$

By distributing the complement pattern p on the subterms, the algorithm builds a set $Q_c(p)$ of tuples $q = (q_1, \ldots, q_n)$ of patterns, with each q_i being either $\bot$ or a subterm of p, such that

$$[\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus p]\!] = \bigcup_{q \in Q_c(p)} [\![c(x_{s_1}^{-p} \setminus q_1, \ldots, x_{s_n}^{-p} \setminus q_n)]\!] \tag{2}$$

We have thus

$$\begin{aligned}
\{\!\{x_s^{-p}\}\!\} &= \{u_{|\omega} \mid u \in [\![x_s^{-p}]\!], \omega \in \mathcal{P}os\,(u)\} \\
&= \left\{ u_{|\omega} \mid u \in \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c(p)} [\![c(x_{s_1}^{-p} \setminus q_1, \ldots, x_{s_n}^{-p} \setminus q_n)]\!], \omega \in \mathcal{P}os\,(u) \right\} \\
&= \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c(p)} \left\{ u_{|\omega} \mid u \in [\![c(x_{s_1}^{-p} \setminus q_1, \ldots, x_{s_n}^{-p} \setminus q_n)]\!], \omega \in \mathcal{P}os\,(u) \right\} \\
&= \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c(p)} \{\!\{c(x_{s_1}^{-p} \setminus q_1, \ldots, x_{s_n}^{-p} \setminus q_n)\}\!\} \text{ (def. of deep semantics)} \qquad (3) \\
&= \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q'_c(p)} [\![c(x_{s_1}^{-p} \setminus q_1, \ldots, x_{s_n}^{-p} \setminus q_n)]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q'_c(p)} \bigcup_{i=1}^{n} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\} \\
&= \qquad\qquad [\![x_s^{-p}]\!] \qquad\qquad \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q'_c(p)} \bigcup_{i=1}^{n} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}
\end{aligned}$$

with $Q'_c(p) \subseteq Q_c(p)$ s.t. $\forall q = (q_1, \ldots, q_n) \in Q'_c(p), [\![x_{s_i}^{-p} \setminus q_i]\!] \neq \emptyset, i \in [1,n]$.

Note that x_s^{-p} is the same as $x_s^{-p} \setminus \bot$ and thus, in order to express the deep semantics of annotated variables as a union of ground semantics the algorithm computes a fixpoint for the equation

$$\{\!\{x_s^{-p} \setminus r\}\!\} = [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q'_c(r+p)} \bigcup_{i=1}^{n} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}$$

Proposition 4.3 (Correctness). *Given $s \in \mathcal{S}, p \in \mathcal{T}_\bot(\mathcal{C}, \mathcal{X})$ and $r : s$ a sum of constructor patterns, $\mathsf{getReachable}(s, p, \emptyset, r)$ terminates and if we have $R = \mathsf{getReachable}(s, p, \emptyset, r)$, then*

$$\{\!\{x_s^{-p} \setminus r\}\!\} = \bigcup_{(s', p') \in R} [\![x_{s'}^{-p} \setminus p']\!]$$

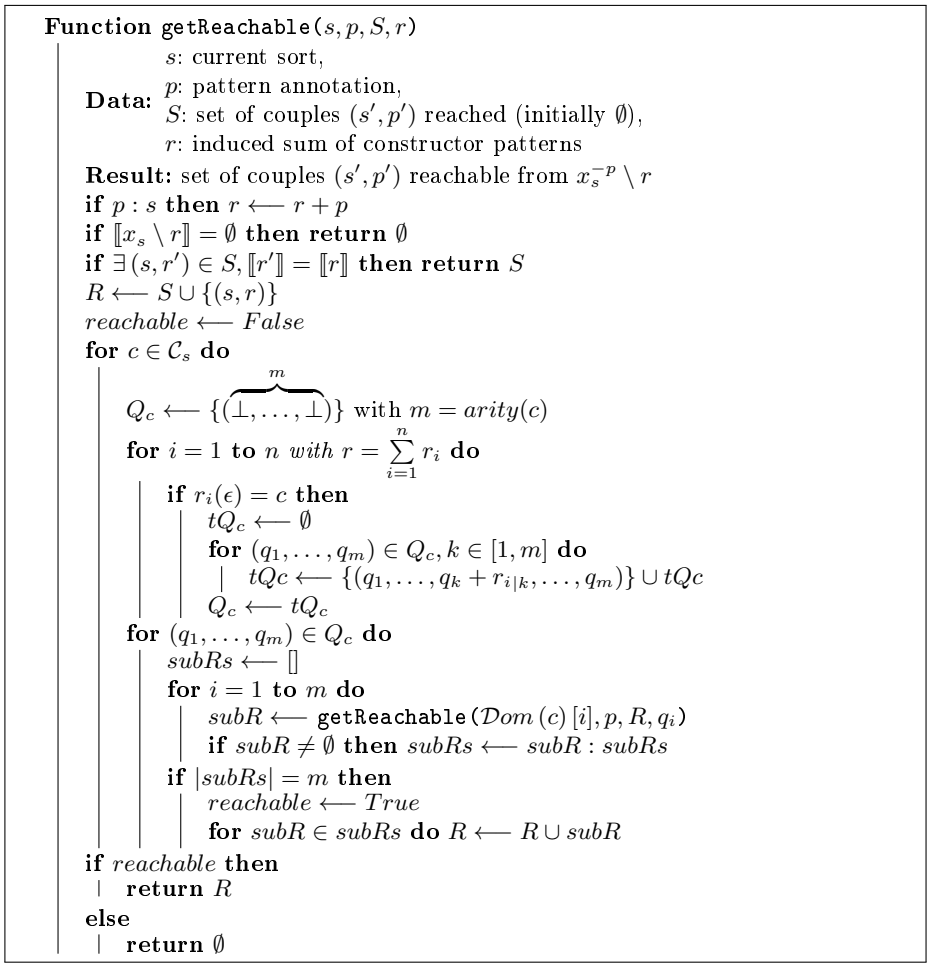

Fig. 1. Compute the deep semantics of quasi-additive patterns as a union of ground semantics. The boolean *reachable* indicates if we can exhibit at least one p-free value headed by one of the constructors of s. The set Q_c corresponds to $Q_c(r)$ in Equation 2 and is built by accumulation of the pattern complements from r for the arguments of a pattern headed by c. Given a tuple $q \in Q_c$, *subRs* is a list (built with :) which stores the recursive results of getReachable over each element of q.

Moreover, we have $[\![x_s^{-p} \setminus r]\!] = \emptyset$ *iff* $R = \emptyset$.

Example 4.1. We consider the symbolic patterns from Example 3.2 and express their deep semantics as explained above. According to Proposition 4.2, we have $[\![\tilde{r_1}]\!] = [\![c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})]\!] = [\![c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})]\!] \cup [\![x_{s_2}^{-p_2}]\!] \cup [\![y_{s_1}^{-p_1}]\!]$ and we should expand $[\![x_{s_2}^{-p_2}]\!]$ and $[\![y_{s_1}^{-p_1}]\!]$.

To expand $[\![y_{s_1}^{-p_1}]\!]$ the sets $Q_c(p_1)$ are computed for each $c \in \mathcal{C}_{s_1} = \{c_1, c_2\}$. First, following equation (1), $[\![y_{s_1}^{-p_1}]\!] = [\![c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1}) \setminus c_1(c_4(z_{s_3}^{-\bot}), y_{s_1}^{-\bot})]\!] \cup$

$[\![c_2(z_{s_3}^{-p_1}) \setminus c_1(c_4(z_{s_3}^{-\perp}), y_{s_1}^{-\perp})]\!]$ and we can easily see that the complement relation in terms of ground semantics corresponds to set differences of cartesian products: $[\![c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1}) \setminus c_1(c_4(z_{s_3}^{-\perp}), y_{s_1}^{-\perp})]\!] = [\![c_1(x_{s_2}^{-p_1} \setminus c_4(z_{s_3}^{-\perp}), y_{s_1}^{-p_1})]\!] \cup [\![c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1} \setminus y_{s_1}^{-\perp})]\!]$. We get thus, $[\![y_{s_1}^{-p_1}]\!] = [\![c_1(x_{s_2}^{-p_1} \setminus c_4(z_{s_3}^{-\perp}), y_{s_1}^{-p_1})]\!] \cup [\![c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1} \setminus y_{s_1}^{-\perp})]\!] \cup [\![c_2(z_{s_3}^{-p_1})]\!]$. Hence, following equation (2), $Q_{c_1}(p_1) = \{(c_4(z_{s_3}^{-\perp}), \perp), (\perp, y_{s_1})\} = \{(p_2, \perp), (\perp, y_{s_1})\}$ and $Q_{c_2}(p_1) = \{(\perp)\}$. Moreover, $[\![c_1(x_{s_2}^{-p_1} \setminus p_2, y_{s_1}^{-p_1})]\!]$ and $[\![c_2(z_{s_3}^{-p_1})]\!]$ are not empty (since $c_1(c_3(c_2(c_6)), c_2(c_6))$ and $c_2(c_6)$ belong respectively to each of them) while $[\![c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1} \setminus y_{s_1}^{-\perp})]\!]$ is clearly empty. Thus,

$$\{\!\{y_{s_1}^{-p_1}\}\!\} = [\![y_{s_1}^{-p_1}]\!] \cup \{\!\{x_{s_2}^{-p_1} \setminus p_2\}\!\} \cup \{\!\{y_{s_1}^{-p_1}\}\!\} \cup \{\!\{z_{s_3}^{-p_1}\}\!\}.$$

The `getReachable` algorithm continues the expansions until a fixpoint is reached. More precisely, we get $\{\!\{y_{s_1}^{-p_1}\}\!\} = [\![y_{s_1}^{-p_1}]\!] \cup [\![z_{s_3}^{-p_1}]\!] \cup [\![x_{s_2}^{-p_1} \setminus p_2]\!]$ and $\{\!\{x_{s_2}^{-p_2}\}\!\} = [\![x_{s_2}^{-p_2}]\!] \cup [\![y_{s_1}^{-p_2}]\!] \cup [\![z_{s_3}^{-p_2}]\!]$, and therefore, the deep semantics of $\tilde{r_1} = c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})$ is the union of $[\![c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})]\!]$, $[\![y_{s_1}^{-p_1}]\!]$, $[\![z_{s_3}^{-p_1}]\!]$, $[\![x_{s_2}^{-p_1} \setminus p_2]\!]$, $[\![x_{s_2}^{-p_2}]\!]$, $[\![y_{s_1}^{-p_2}]\!]$ and $[\![z_{s_3}^{-p_2}]\!]$.

Propositions 4.2 and 4.3 guarantee that the deep semantics of any symbolic pattern and thus, of any term, can actually be expressed as the union of ground semantics of quasi-additive patterns. We introduce in the next section a method to automatically verify that the corresponding intersections with the semantics of a given pattern p are empty and check thus that a term is p-free.

4.2 Establishing pattern-free properties

Compared to the approach proposed in [9], we have to provide a method that also takes into account the specific behaviour of annotated variables. On the other hand, in order to establish pattern-free properties, we only need to check that the intersection of the semantics of a symbolic pattern t with the semantics of the given constructor pattern p is empty: thus, we want a TRS that reduces a pattern of the form $t \times p$ to $\perp$ if and only if its ground semantics is empty.

To this end, we introduce the TRS $\mathcal{R}_p$ presented in Figure 2. 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 corresponding set operations.

The rules A1, A2, resp. E2, E3, describe the behaviour of the conjunction, resp. the disjunction, *w.r.t.* $\perp$. Rule E1 indicates that the semantics of a pattern containing a subterm with an empty ground semantics is itself empty, while rule S1 corresponds to the distributivity of conjunction over cartesian products. Similarly, rules S2 and S3 express the distributivity of conjunction over 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 operation laws for complements. Rule M7 corresponds to the set difference of cartesian products; the case when the head symbol is a constant c corresponds to the rule $c \setminus c \Rightarrow \perp$. Rule M8 corresponds to the special case where complemented sets are disjoint.

Remove empty sets:

(A1) $\bot + \overline{v} \Rightarrow \overline{v}$

(A2) $\overline{v} + \bot \Rightarrow \overline{v}$

Distribute sets:

(E1) $\delta(\overline{v_1}, \ldots, \bot, \ldots, \overline{v_n}) \Rightarrow \bot$

(E2) $\bot \times \overline{v} \Rightarrow \bot$

(E3) $\overline{v} \times \bot \Rightarrow \bot$

(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) $(\overline{w_1} + \overline{w_2}) \times \overline{v} \Rightarrow (\overline{w_1} \times \overline{v}) + (\overline{w_2} \times \overline{v})$

(S3) $\overline{w} \times (\overline{v_1} + \overline{v_2}) \Rightarrow (\overline{w} \times \overline{v_1}) + (\overline{w} \times \overline{v_2})$

Simplify complements:

(M1) $\overline{v} \setminus \overline{x_s}^{\bot} \Rightarrow \bot$

(M2) $\overline{v} \setminus \bot \Rightarrow \overline{v}$

(M3) $(\overline{v_1} + \overline{v_2}) \setminus \overline{w} \Rightarrow (\overline{v_1} \setminus \overline{w}) + (\overline{v_2} \setminus \overline{w})$

(M5) $\bot \setminus \overline{v} \Rightarrow \bot$

(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) $\overline{v} \times \overline{x_s}^{\bot} \Rightarrow \overline{v}$

(T2) $\overline{x_s}^{\bot} \times \overline{v} \Rightarrow \overline{v}$

(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) $\alpha(\overline{v_1}, \ldots, \overline{v_n}) \times \beta(\overline{w_1}, \ldots, \overline{w_m}) \Rightarrow \bot$ *with* $\alpha \neq \beta$

Simplify p-free:

(P1) $\overline{x_s}^{\overline{p}} \times \alpha(\overline{v_1}, \ldots, \overline{v_n}) \Rightarrow \sum_{c \in \mathcal{C}_s} c(z_1{}_{s_1}^{-p}, \ldots, z_m{}_{s_m}^{-p}) \times (\alpha(\overline{v_1}, \ldots, \overline{v_n}) \setminus \overline{p})$ *with* $m = arity(c)$

(P2) $\alpha(\overline{v_1}, \ldots, \overline{v_n}) \times (\overline{x_s}^{\overline{p}} \setminus \overline{t}) \Rightarrow (\alpha(\overline{v_1}, \ldots, \overline{v_n}) \times \overline{x_s}^{\overline{p}}) \setminus \overline{t}$ if $[\![\overline{x_s}^{\overline{p}} \setminus \overline{t}]\!] \neq \emptyset$

(P3) $\overline{x_s}^{\overline{q}} \times (\overline{x_s}^{\overline{p}} \setminus \overline{t}) \Rightarrow (\overline{x_s}^{\overline{q}} \times \overline{x_s}^{\overline{p}}) \setminus \overline{t}$ if $[\![\overline{x_s}^{\overline{p}} \setminus \overline{t}]\!] \neq \emptyset$

(P4) $(\overline{x_s}^{\overline{p}} \setminus \overline{t}) \times \overline{v} \Rightarrow (\overline{x_s}^{\overline{p}} \times \overline{v}) \setminus \overline{t}$ if $[\![\overline{x_s}^{\overline{p}} \setminus \overline{t}]\!] \neq \emptyset$

(P5) $(\overline{x_s}^{\overline{p}} \setminus \overline{t}) \setminus \overline{u} \Rightarrow \overline{x_s}^{\overline{p}} \setminus (\overline{t} + \overline{u})$ if $[\![\overline{x_s}^{\overline{p}} \setminus \overline{t}]\!] \neq \emptyset$

(P6) $\overline{x_s}^{\overline{p}} \setminus \overline{t} \Rightarrow \bot$ if $[\![\overline{x_s}^{\overline{p}} \setminus \overline{t}]\!] = \emptyset$

Fig. 2. $\mathcal{R}_p$: reduce pattern of the form $t \times p$; $\overline{v}, \overline{v_1}, \ldots, \overline{v_n}$, $\overline{w}, \overline{w_1}, \ldots, \overline{w_n}$ range over quasi-additive patterns, $\overline{u}, \overline{t}$ range over pure regular additive patterns, $\overline{t_1}, \ldots, \overline{t_n}$ range over pure symbolic patterns, $\overline{p}, \overline{q}$ range over constructor patterns, $\overline{x}$ ranges over pattern variables. α, β expand to all the symbols in $\mathcal{C}$, δ expands to all symbols in $\mathcal{C}^{n>0}$.

The rules T1 and T2 indicate that the intersection with the set of all terms has no effect, rule T3 corresponds to distribution laws for the joint intersection, while T4 corresponds to the disjointed case.

We have seen that the ground semantics of an annotated variable is obtained by considering, for each constructor of the appropriate sort, the set of all terms having this symbol at the root position complemented by the pattern in the annotation and taking the union of all these sets. $\mathcal{R}_p$ uses this property in the rule P1 to expand annotated variables allowing thus for the triggering of the other rules for conjunction. Note that z_i are fresh variables generated automatically. The rules P2, P3 and P4 express the respective behaviour of conjunction over complements ($A \cap (B \setminus C) = (A \setminus C) \cap B = (A \cap B) \setminus C$).

Finally, we can observe that, thanks to the algorithm introduced in Figure 1, we can determine if $\{\!\{\overline{x}_s^{-\overline{p}} \setminus \overline{v}\}\!\} = \emptyset$. Moreover, by definition, $\{\!\{t\}\!\} = \emptyset$ if and only if $[\![t]\!] = \emptyset$. Therefore, the TRS is finalized by the rule P6 which eliminates (when possible) annotated variables. In order to apply P6 exhaustively, $\mathcal{R}_p$ also needs a rule to perform some \-factorization around variables, resulting in the rule P5.

Proposition 4.4 (Semantics preservation). *For any extended patterns p, q, if $p \twoheadrightarrow_{\mathcal{R}_p} q$ then $[\![p]\!] = [\![q]\!]$.*

We have seen that the algorithm in Figure 1 always terminates and that it can be used to decide the conditions in the TRS $\mathcal{R}_p$ (Proposition 4.3). Based on this, we can prove the convergence of the TRS $\mathcal{R}_p$. While we cannot provide a simple description of the normal forms obtained by reduction of a general extended pattern, $\mathcal{R}_p$ can be used to establish the emptiness of a given intersection:

Proposition 4.5. *The rewriting system $\mathcal{R}_p$ is confluent and terminating. Given a quasi-additive pattern t and a constructor pattern p, we have $t \times p \twoheadrightarrow_{\mathcal{R}_p} \bot$ if and only if $[\![t \times p]\!] = \emptyset$.*

4.3 Establishing semantics preserving properties

The approach proposed in the previous section allows the systematic verification of pattern-free properties for any term in $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $\tilde{t}$ is linear. It is easy to see if we denote by $L(t)$ the term obtain by replacing all the variables in the term t by fresh ones then $[\![t]\!] \subseteq [\![L(t)]\!]$. We can thus linearize, if necessary, the right-hand sides of the rules of a CBTRS and subsequently check that it is semantics preserving.

Example 4.2. We apply the approach to check that the CBTRS in Example 3.3 is semantics preserving. For this we need to prove that $c_1(g^{-p_2}(x_{s_2}), f^{-p_1}(y_{s_1}))$ and $c_2(z_{s_3})$ are p_1-free, and that $c_3(c_2(z_{s_3}))$ and $c_3(f^{-p_1}(y_{s_1}))$ are p_2-free.

In order to prove that $r_1 = c_1(g^{-p_2}(x_{s_2}), f^{-p_1}(y_{s_1}))$ is p_1-free, we should first compute the deep semantics of $\tilde{r_1} = c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})$ and we have seen in Example 4.1 how `getReachable` is used to compute this deep semantics as the union of $[\![c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})]\!]$, $[\![y_{s_1}^{-p_1}]\!]$, $[\![z_{s_3}^{-p_1}]\!]$, $[\![x_{s_2}^{-p_1} \setminus p_2]\!]$, $[\![x_{s_2}^{-p_2}]\!]$, $[\![y_{s_1}^{-p_2}]\!]$ and $[\![z_{s_3}^{-p_2}]\!]$. For all the terms in the union we compute their conjunction with p_1 using $\mathcal{R}_p$ which reduces them all to $\bot$. Hence, by Proposition 4.1, r_1 is p_1-free.

Similarly, we can check that $c_2(z_{s_3})$ is p_1-free, and $c_3(c_2(z_{s_3}))$ and $c_3(f(y_{s_1}))$ are p_2-free. Thus, the CBTRS is semantics preserving. It is easy to check that it is also terminating and consequently, the normal form of any term $f(t)$, $t \in \mathcal{T}_{s_1}(\mathcal{F})$, is p_1-free and the normal form of any term $g(u)$, $u \in \mathcal{T}_{s_2}(\mathcal{F})$, is p_2-free.

We can now come back to the initial flattening example presented in the introduction. We consider a signature consisting of the sorts and constructors already presented in the introduction to which we add the defined symbols $\mathcal{D} = \{flattenE^{-p} : Expr \mapsto Expr, flattenL^{-p} : List \mapsto List, concat^{-\bot} : List \times$

$List \mapsto List\}$. with $p = cons(lst(l_1), l_2)$, to indicate that the corresponding functions defined by the following CBTRS aim at eliminating this pattern:

$$\begin{cases} flattenE^{-p}(str(s)) & \to str(s) \\ flattenE^{-p}(lst(l)) & \to lst(flattenL^{-p}(l)) \\ flattenL^{-p}(nil) & \to nil \\ flattenL^{-p}(cons(str(s), l)) & \to cons(str(s), flattenL^{-p}(l)) \\ flattenL^{-p}(cons(lst(l_1), l_2)) & \to flattenL^{-p}(concat^{-\bot}(l_1, l_2)) \\ concat^{-\bot}(cons(e, l_1), l_2) & \to cons(e, concat^{-\bot}(l_1, l_2)) \\ concat^{-\bot}(nil, l) & \to l \end{cases}$$

Thanks to the method introduced in the previous section we can check that the right-hand sides of the first 5 rules are p-free and hence, as explained in Section 3.3, that the CBTRS is semantics preserving. This CBTRS is clearly terminating and complete and thus, we can guarantee that the normal forms of terms headed by $flattenE$ or $flattenL$ are p-free values.

The method has been implemented in Haskell[3]. The implementation takes as input a file defining the signature and the CBTRS to be checked and returns the (potentially empty) set of non pattern-free preserving rules (*i.e.* rules that do not satisfy the pattern-free requirements implied by the signature). For each such rule we provide a set of terms whose ground semantics is included in the deep semantics of the right-hand side of the rule and that do not satisfy the pattern-free property required by the left-hand side.

The complexity of the method for checking the pattern-freeness *w.r.t.* to a given pattern p is exponential on the depth of p with a growth rate proportional to the (maximum) arity of the symbols present in p. Benchmarks performed on the implementation optimized to minimize repetitive computations showed that, when considering terms and patterns of depth 5 with symbols of arity 6, checking the pattern-freeness of a single term takes $\sim$ 200ms, and checking the semantics preservation of a CBTRS of 25 rules takes $\sim$ 3s (on an Intel Core i5-8250U). In practice, the size of the pattern annotations is generally lower that the ones we experimented with and we consider that despite the exponential complexity the concrete performances are reasonable for a static analysis technique.

5 Related work

While the work presented in this paper introduces an original approach to express and ensure a particular category of syntactical guarantees associated to program transformation, a number of different approaches presenting methods to obtain some guarantees for similar classes of functions exist in the literature.

Tree automata completion Tree automata completion consists in techniques used to compute an approximation of the set of terms reachable by a rewriting relation [14]. Such techniques could, therefore, be applied to solve similar problems

[3] the source code 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`.

to the one presented in this paper. The application of this approach is nevertheless usually conditioned by the termination of both the TRS and the set of equational approximations used [31,15]. Thus, while providing sometimes a more precise characterization of the approximations of the normal forms, these techniques are constrained, in terms of termination, by some syntactical conditions. When testing 5 of our base case scenarios with two popular implementations, Timbuk3 [14] seems less powerful than our approach, while Timbuk4 [18] can check more systems but less efficiently than our approach. For Timbuk3, the over-approximation strategies were to broad to check all considered examples. Nonetheless, it was able to check the properties using exact normalization for 2 of these examples. This was, for example, the case for a rewritten version of the flattening TRS which avoided the nested function calls, in order to build a TRS of a known terminating class for automata completion without approximation [15]. Timbuk4, recently proposed to use a counter-example based abstraction refinement procedure to control the over-approximation [18], could check all the examples including a version of the flattening TRS which could not be verified with our current approach. On the other hand, the computational performance is considerably worse than for our approach ($\sim$ 700ms for the flatten case compared to $\sim 20\mu$s for our approach). Moreover, for Timbuk3 and Timbuk4, the target CBTRS has to be extended with a function encoding the desired pattern-freeness property in order to check it.

Recursion schemes Some formalisms propose to deal with higher order functions through the use of higher order recursion schemes, a form of higher order grammars that are used as generators of (possibly infinite) trees [23]. 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 [28] and provide the basis for automatic abstraction refinement. These techniques address in a clever way the control-flow analysis of functional programs while the formalism proposed in our work is more focused on providing syntactic guarantees on the shape of the tree obtained through a pass-like transformation. The use of the annotation system also contributes to a more precise way to express and control the considered over-approximation.

Tree transducers Besides term rewriting systems, another popular approach for specifying transformations consists in the use of tree transducers [24]. Transducers have indeed been shown to have a number of appealing properties when applied for strings, even infinite [2], and most notably can provide an interesting approach for model checking certain classes of programs thanks to the decidability of general verification problems [1]. Though the verification problems we tackle here are significantly more strenuous for tree transducers, Kobayashi et al. introduced in [24] a class of higher order tree transducers which can be modeled by recursion schemes and thus, provided a sound and complete algorithm to solve verification problems over that class. We claim that annotated CBTRSs are easier to grasp when specifying pass-like transformations and are less intrusive for expressing the pattern-free properties.

Refinement types Formalisms such as refinement types [11] can be seen as an alternative approach for verifying the absence, or presence, of specific patterns. In particular, notions such as constructor subtypes [6] could be used to construct complex type systems whose type checking would provide guarantees similar to the ones provided by our formalism. This would however result in the construction of multiple type systems in order to type check each transformation as was the case in the original inspiration of our work [22].

6 Conclusion and perspectives

We have proposed a method to statically analyse constructor term rewrite systems and verify the absence of patterns from the corresponding normal forms. We can thus guarantee not only that some constructors are not present in the normal forms but we can also be more specific and verify that more complex constructs cannot be retrieved in the result of the reduction. Such an approach avoids the burden of specifying a specific language to characterize the result of each (intermediate) transformation, as the user is simply requested to indicate the patterns that should be eliminated by the respective transformation.

Different termination analysis techniques [3,19,17] and corresponding tools like AProVE [16,12] and TTT2 [25] can be used for checking the termination of the rewriting systems before applying our method for checking pattern-free properties. On the other hand, the approach applies also for CBTRS which are not complete or not strongly normalising and still guarantees that all the intermediate terms in the reduction are pattern-free; in particular, if the CBTRS is weakly normalising the existing normal forms are pattern-free. It is worth mentioning that the approach extends straightforwardly to sums of constructor patterns of the form $p = p_1 + \cdots + p_n$ in the annotations to indicate simultaneously p_i-freeness *w.r.t.* all the patterns in the sum.

We believe this formalism opens a lot of opportunities for further developments. In the current version, the verification relies on an over-approximation of the set of reducts and thus, can lead to false negatives. For example, an alternative rule $flattenL(cons(lst(l_1), l_2)) \twoheadrightarrow concat(flattenL(l_1), flattenL(l_2))$ in our flattening CBTRS would be reported as non pattern-preserving. In our experience, such false negatives arise when the annotations for some symbols are not precise enough in specifying the expected behaviour (*e.g.* the annotations for *concat* do not specify that the concatenation of two flatten lists is supposed to be a flatten lists) and, although we conjecture this might indicate some issues in the design of the CBTRS, we work on an alternative approach allowing for a finer-grain analysis. While false negatives could also arise when the right-hand side of a rule has to be linearized, the current implementation already uses an aliasing technique to handle such cases; the technical details have been omitted in the paper due to the space restrictions.

We also intend to extend and use the approach in the context of automatic rewrite rule generation techniques, such as the one introduced in [8], in order to automatize the generation of boilerplate code as in [22].

References

1. Alur, R., Cerný, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011. pp. 599–610. ACM (2011). https://doi.org/10.1145/1926385.1926454
2. Alur, R., Filiot, E., Trivedi, A.: Regular transformations of infinite strings. In: IEEE Symposium on Logic in Computer Science, LICS 2012. pp. 65–74. IEEE Computer Society (2012). https://doi.org/10.1109/LICS.2012.18
3. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science **236**(1-2), 133–178 (2000). https://doi.org/10.1016/S0304-3975(99)00207-8
4. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
5. Balland, E., Brauner, P., Kopetz, R., Moreau, P., Reilles, A.: Tom: Piggybacking rewriting on Java. In: Term Rewriting and Applications, 18th International Conference, RTA 2007. Lecture Notes in Computer Science, vol. 4533, pp. 36–47. Springer (2007). https://doi.org/10.1007/978-3-540-73449-9_5
6. Barthe, G., Frade, M.J.: Constructor subtyping. In: Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99. Lecture Notes in Computer Science, vol. 1576, pp. 109–127. Springer (1999). https://doi.org/10.1007/3-540-49099-X_8
7. Bellegarde, F.: Program transformation and rewriting. In: Rewriting Techniques and Applications, 4th International Conference, RTA-91. Lecture Notes in Computer Science, vol. 488, pp. 226–239. Springer (1991). https://doi.org/10.1007/3-540-53904-2_99
8. Cirstea, H., Lenglet, S., Moreau, P.: A faithful encoding of programmable strategies into term rewriting systems. In: 26th International Conference on Rewriting Techniques and Applications, RTA 2015. LIPIcs, vol. 36, pp. 74–88. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.74
9. Cirstea, H., Moreau, P.: Generic encodings of constructor rewriting systems. In: International Symposium on Principles and Practice of Programming Languages, PPDP 2019. pp. 8:1–8:12. ACM (2019). https://doi.org/10.1145/3354166.3354173
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: The maude 2.0 system. In: Rewriting Techniques and Applications, 14th International Conference, RTA 2003. Lecture Notes in Computer Science, vol. 2706, pp. 76–87. Springer (2003). https://doi.org/10.1007/3-540-44881-0_7
11. Freeman, T.S., Pfenning, F.: Refinement types for ML. In: ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI). pp. 268–277. ACM (1991). https://doi.org/10.1145/113445.113468
12. Fuhs, C., Giesl, J., Parting, M., Schneider-Kamp, P., Swiderski, S.: Proving termination by dependency pairs and inductive theorem proving. Journal of Automatic Reasoning **47**(2), 133–160 (2011). https://doi.org/10.1007/s10817-010-9215-9
13. Garrigue, J.: Programming with polymorphic variants. In: In ACM Workshop on ML (1998)
14. Genet, T.: Towards static analysis of functional programs using tree automata completion. In: Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014. Lecture Notes in Computer Science, vol. 8663, pp. 147–161. Springer (2014). https://doi.org/10.1007/978-3-319-12904-4_8

15. Genet, T.: Termination criteria for tree automata completion. Journal of Logical and Algebraic Methods in Programming **85**(1), 3–33 (2016). https://doi.org/10.1016/j.jlamp.2015.05.003
16. Giesl, J., Schneider-Kamp, P., Thiemann, R.: Automatic termination proofs in the dependency pair framework. In: Automated Reasoning, Third International Joint Conference, IJCAR 2006. Lecture Notes in Computer Science, vol. 4130, pp. 281–286. Springer (2006). https://doi.org/10.1007/11814771_24
17. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automatic Reasoning **37**(3), 155–203 (2006). https://doi.org/10.1007/s10817-006-9057-7
18. Haudebourg, T., Genet, T., Jensen, T.P.: Regular language type inference with term rewriting. ACM on Programming Languages **4**(ICFP), 112:1–112:29 (2020). https://doi.org/10.1145/3408994
19. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Information and Computation **199**(1-2), 172–199 (2005). https://doi.org/10.1016/j.ic.2004.10.004
20. Jouannaud, J., Kirchner, C.: Solving equations in abstract algebras: A rule-based survey of unification. In: Computational Logic - Essays in Honor of Alan Robinson, pp. 257–321. The MIT Press (1991)
21. Jouannaud, J., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM Journal on Computing **15**(4), 1155–1194 (1986). https://doi.org/10.1137/0215084
22. Keep, A.W., Dybvig, R.K.: A nanopass framework for commercial compiler development. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'13. pp. 343–350. ACM (2013). https://doi.org/10.1145/2500365.2500618
23. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 20099. pp. 416–428. ACM (2009). https://doi.org/10.1145/1480881.1480933
24. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010. pp. 495–508. ACM (2010). https://doi.org/10.1145/1706299.1706355
25. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Treinen, R. (ed.) Rewriting Techniques and Applications, 20th International Conference, RTA 2009. Lecture Notes in Computer Science, vol. 5595, pp. 295–304. Springer (2009). https://doi.org/10.1007/978-3-642-02348-4_21
26. Lacey, D., de Moor, O.: Imperative program transformation by rewriting. In: Wilhelm, R. (ed.) Compiler Construction, 10th International Conference, CC 2001. Lecture Notes in Computer Science, vol. 2027, pp. 52–68. Springer (2001). https://doi.org/10.1007/3-540-45306-7_5
27. Meseguer, J., Braga, C.: Modular rewriting semantics of programming languages. In: Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Lecture Notes in Computer Science, vol. 3116, pp. 364–378. Springer (2004). https://doi.org/10.1007/978-3-540-27815-3_29
28. Ong, C.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011. pp. 587–598. ACM (2011). https://doi.org/10.1145/1926385.1926453

29. Pottier, F.: Visitors unchained. ACM on Programming Languages **1**(ICFP), 28:1–28:28 (2017). https://doi.org/10.1145/3110272
30. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
31. Takai, T.: A verification technique using term rewriting systems and abstract interpretation. In: Rewriting Techniques and Applications, 15th International Conference, RTA 2004. Lecture Notes in Computer Science, vol. 3091, pp. 119–133. Springer (2004). https://doi.org/10.1007/978-3-540-25979-4_9
32. Terese: Term Rewriting Systems. Cambridge University Press (2003), m. Bezem, J. W. Klop and R. de Vrijer, eds.
33. Visser, E.: Strategic pattern matching. In: Rewriting Techniques and Applications, 10th International Conference, RTA-99. Lecture Notes in Computer Science, vol. 1631, pp. 30–44. Springer (1999). https://doi.org/10.1007/3-540-48685-2_3

A Comparison with Timbuk (3 and 4)

We define several functions and try to check the corresponding pattern-freeness constraints using both our approach and Timbuk (v3.2 and v4). The constraints are specified using annotations on the defined symbols in our approach and using some auxiliary rules in Timbuk. The complete definitions are presented in the following sections. The corresponding benchmarks are presented in the following table: ✓ indicates that the pattern-freeness conditions could be checked, ✗ indicates that the property couldn't be checked. We also provide the computation time needed (on an Intel Core i5-8250U) to obtain each result (an ∞ computation time means the computation timed out at $200s$).

	pfree check		Timbuk 3.2		Timbuk 4	
flatten1	✓	$21\mu s$	✗	∞	✓	$685ms$
flatten2	✗	$31\mu s$	✗	∞	✓	$975ms$
flatten3	✓	$36\mu s$	✓	$1.6ms$	✓	$1,4s$
negativeNF	✓	$395\mu s$	✓	$3.2ms$	✓	$104s$
skolemization	✓	$45\mu s$	✗	$1,5s$	✓	$1,6s$

A.1 Flattening functions: flatten1, flatten2, flatten3

We consider the signature consisting of the following sorts and constructor symbols

$$\begin{aligned} Expr &= str(String) & List &= nil \\ &\mid lst(List) & &\mid cons(Expr, List) \end{aligned}$$

and define several versions for a function whose purpose is to flatten lists as explained in Section 4.3.

First version: flatten1
The first version of the flattening function is defined using the following (annotated) defined symbols:

$$\begin{aligned} \boldsymbol{flattenE}^{-cons(lst(l_1),l_2)} &: Expr \mapsto Expr \\ \boldsymbol{flattenL}^{-cons(lst(l_1),l_2)} &: List \mapsto List \\ \boldsymbol{concat} &: List \times List \mapsto List \end{aligned}$$

and the CBTRS:

$$\begin{cases} \boldsymbol{flattenE}(str(s)) & \to str(s) \\ \boldsymbol{flattenE}(lst(l)) & \to lst(\boldsymbol{flattenL}(l)) \\ \boldsymbol{flattenL}(nil) & \to nil \\ \boldsymbol{flattenL}(cons(str(s), l)) & \to cons(str(s), \boldsymbol{flattenL}(l)) \\ \boldsymbol{flattenL}(cons(lst(l_1), l_2)) & \to \boldsymbol{flattenL}(\boldsymbol{concat}(l_1, l_2)) \\ \boldsymbol{concat}(cons(e, l_1), l_2) & \to cons(e, \boldsymbol{concat}(l_1, l_2)) \\ \boldsymbol{concat}(nil, l) & \to l \end{cases}$$

In Timbuk, the following rules are added to the CBTRS to encode and verify the desired properties expressed as annotations above:

$$\begin{cases} \boldsymbol{isflat}(cons(lst(X), Y)) & \to false \\ \boldsymbol{isflat}(cons(A, Y)) & \to \boldsymbol{isflat}(Y) \\ \boldsymbol{isflat}(cons(B, Y)) & \to \boldsymbol{isflat}(Y) \\ \boldsymbol{isflat}(nil) & \to true \end{cases}$$

Second version: flatten2

The second version of the flattening function is defined using the same (annotated) defined symbols as before:

$$\begin{array}{ll} \boldsymbol{flattenE}^{-cons(lst(l_1),l_2)} & : Expr \mapsto Expr \\ \boldsymbol{flattenL}^{-cons(lst(l_1),l_2)} & : List \mapsto List \\ \boldsymbol{concat} & : List \times List \mapsto List \end{array}$$

and the CBTRS:

$$\begin{cases} \boldsymbol{flattenE}(str(s)) & \to str(s) \\ \boldsymbol{flattenE}(lst(l)) & \to lst(\boldsymbol{flattenL}(l)) \\ \boldsymbol{flattenL}(nil) & \to nil \\ \boldsymbol{flattenL}(cons(str(s), l)) & \to cons(str(s), \boldsymbol{flattenL}(l)) \\ \boldsymbol{flattenL}(cons(lst(l_1), l_2)) & \to \boldsymbol{concat}(\boldsymbol{flattenL}(l_1), \boldsymbol{flattenL}(l_2)) \\ \boldsymbol{concat}(cons(e, l_1), l_2) & \to cons(e, \boldsymbol{concat}(l_1, l_2)) \\ \boldsymbol{concat}(nil, l) & \to l \end{cases}$$

For Timbuk, the same rules as before are added to the CBTRS to encode and verify the desired properties expressed as annotations in our approach.

Third version: flatten3

The third version of the flattening function is defined using the following (annotated) defined symbols:

$$\begin{array}{ll} \boldsymbol{flattenE}^{-cons(lst(l_1),l_2)} & : Expr \mapsto Expr \\ \boldsymbol{flattenL}^{-cons(lst(l_1),l_2)} & : List \mapsto List \\ \boldsymbol{fconcat}^{-cons(lst(l_1),l_2)} & : List \times List \mapsto List \end{array}$$

and the CBTRS:

$$\begin{cases} \boldsymbol{flattenE}(str(s)) & \to str(s) \\ \boldsymbol{flattenE}(lst(l)) & \to lst(\boldsymbol{flattenL}(l)) \\ \boldsymbol{flattenL}(nil) & \to nil \\ \boldsymbol{flattenL}(cons(str(s), l)) & \to cons(str(s), \boldsymbol{flattenL}(l)) \\ \boldsymbol{flattenL}(cons(lst(l_1), l_2)) & \to \boldsymbol{fconcat}(l_1, l_2) \\ \boldsymbol{fconcat}(cons(str(s), l_1), l_2) & \to cons(str(s), \boldsymbol{fconcat}(l_1, l_2)) \\ \boldsymbol{fconcat}(cons(lst(l_1), l_2), l_3) & \to \boldsymbol{fconcat}(l_1, cons(lst(l_2), l_3)) \\ \boldsymbol{fconcat}(nil, l) & \to \boldsymbol{flattenL}(l) \end{cases}$$

For Timbuk, the same rules as before are added to the CBTRS to encode and verify the desired properties expressed as annotations in our approach.

A.2 Negative normal forms: negativeNF

We consider a signature consisting of the following sorts and constructor symbols

$$\begin{array}{ll} Formula = Predicate(String) & \mid Impl(Formula, Formula) \\ \quad \mid Not(Formula) & \mid Exists(String, Formula) \\ \quad \mid And(Formula, Formula) & \mid Forall(String, Formula) \\ \quad \mid Or(Formula, Formula) & \end{array}$$

and we define a function whose purpose is to compute the negative normal form of a formula using the following (annotated) defined symbols:

$$\boldsymbol{negativeNF}^{-(Impl(e)+Not(!Predicate(s)))} : Formula \mapsto Formula$$

and the CBTRS:

$$\left\{\begin{array}{ll} \boldsymbol{negativeNF}(Predicate(s,t)) & \twoheadrightarrow Predicate(s,t) \\ \boldsymbol{negativeNF}(Not(Predicate(s,t))) & \twoheadrightarrow Not(Predicate(s,t)) \\ \boldsymbol{negativeNF}(Not(And(p1,p2))) & \twoheadrightarrow Or(\boldsymbol{negativeNF}(Not(p1)), \boldsymbol{negativeNF}(Not(p2))) \\ \boldsymbol{negativeNF}(Not(Or(p1,p2))) & \twoheadrightarrow And(\boldsymbol{negativeNF}(Not(p1)), \boldsymbol{negativeNF}(Not(p2))) \\ \boldsymbol{negativeNF}(Not(Exists(s,p))) & \twoheadrightarrow Forall(s, \boldsymbol{negativeNF}(Not(p))) \\ \boldsymbol{negativeNF}(Not(Forall(s,p))) & \twoheadrightarrow Exists(s, \boldsymbol{negativeNF}(Not(p))) \\ \boldsymbol{negativeNF}(Not(Not(p))) & \twoheadrightarrow \boldsymbol{negativeNF}(p) \\ \boldsymbol{negativeNF}(Not(Impl(p1,p2))) & \twoheadrightarrow And(\boldsymbol{negativeNF}(p1), \boldsymbol{negativeNF}(Not(p2))) \\ \boldsymbol{negativeNF}(Impl(p1,p2)) & \twoheadrightarrow Or(\boldsymbol{negativeNF}(Not(p1)), \boldsymbol{negativeNF}(p2)) \\ \boldsymbol{negativeNF}(And(p1,p2)) & \twoheadrightarrow And(\boldsymbol{negativeNF}(p1), \boldsymbol{negativeNF}(p2)) \\ \boldsymbol{negativeNF}(Or(p1,p2)) & \twoheadrightarrow Or(\boldsymbol{negativeNF}(p1), \boldsymbol{negativeNF}(p2)) \\ \boldsymbol{negativeNF}(Exists(s,p)) & \twoheadrightarrow Exists(s, \boldsymbol{negativeNF}(p)) \\ \boldsymbol{negativeNF}(Forall(s,p)) & \twoheadrightarrow Forall(s, \boldsymbol{negativeNF}(p)) \end{array}\right.$$

In Timbuk, the following rules are added to the CBTRS to encode and verify the desired properties expressed as annotations:

$$\left\{\begin{array}{ll} \boldsymbol{isnnf}(Predicate(S,X)) & \twoheadrightarrow true \\ \boldsymbol{isnnf}(Not(Predicate(S,X))) & \twoheadrightarrow true \\ \boldsymbol{isnnf}(Not(And(X,Y))) & \twoheadrightarrow false \\ \boldsymbol{isnnf}(Not(Or(X,Y))) & \twoheadrightarrow false \\ \boldsymbol{isnnf}(Not(Exists(S,X))) & \twoheadrightarrow false \\ \boldsymbol{isnnf}(Not(Forall(S,X))) & \twoheadrightarrow false \\ \boldsymbol{isnnf}(Not(Not(X))) & \twoheadrightarrow false \\ \boldsymbol{isnnf}(Not(Impl(X,Y))) & \twoheadrightarrow false \\ \boldsymbol{isnnf}(Impl(X,Y)) & \twoheadrightarrow false \\ \boldsymbol{isnnf}(Not(Impl(X,Y))) & \twoheadrightarrow false \\ \boldsymbol{isnnf}(And(X,Y)) & \twoheadrightarrow \boldsymbol{bAnd}(\boldsymbol{isnnf}(X), \boldsymbol{isnnf}(Y)) \\ \boldsymbol{isnnf}(Or(X,Y)) & \twoheadrightarrow \boldsymbol{bAnd}(\boldsymbol{isnnf}(X), \boldsymbol{isnnf}(Y)) \\ \boldsymbol{isnnf}(Exists(S,X)) & \twoheadrightarrow \boldsymbol{isnnf}(X) \\ \boldsymbol{isnnf}(Forall(S,X)) & \twoheadrightarrow \boldsymbol{isnnf}(X) \\ \boldsymbol{bAnd}(true, true) & \twoheadrightarrow true \\ \boldsymbol{bAnd}(X, false) & \twoheadrightarrow false \\ \boldsymbol{bAnd}(false, X) & \twoheadrightarrow false \end{array}\right.$$

A.3 Skolemization: skolemization

We consider a signature consisting of the following sorts and constructor symbols

$$\begin{array}{rl} Formula = & Predicate(String, VarList) \\ \mid & And(Formula, Formula) \\ \mid & Or(Formula, Formula) \\ \mid & Exists(String, Formula) \\ \mid & Forall(String, Formula) \end{array} \qquad \begin{array}{rl} Var = & Var(String) \\ \mid & Skolem(String, VarList) \end{array}$$

$$\begin{array}{rl} VarList = & Nil \\ \mid & Cons(Var, VarList) \end{array}$$

and we define a function whose purpose is to perform skolemization of a quantified formula using the following (annotated) defined symbols:

$$\begin{array}{ll} \boldsymbol{skolem}^{-Exists(s,p)} & : Formula \times VarList \mapsto Formula \\ \boldsymbol{replaceVar} & : Formula \times Var \mapsto Formula \\ \boldsymbol{replaceVar} & : VarList \times Var \mapsto VarList \end{array}$$

and the CBTRS:

$$\left\{\begin{array}{ll} \boldsymbol{skolem}(Predicate(s, xs), l) & \twoheadrightarrow Predicate(s, xs) \\ \boldsymbol{skolem}(And(p_1, p_2), l) & \twoheadrightarrow And(\boldsymbol{skolem}(p_1, l), \boldsymbol{skolem}(p_2, l)) \\ \boldsymbol{skolem}(Or(p_1, p_2), l) & \twoheadrightarrow Or(\boldsymbol{skolem}(p_1, l), \boldsymbol{skolem}(p_2, l)) \\ \boldsymbol{skolem}(Exists(x, p), l) & \twoheadrightarrow \boldsymbol{skolem}(\boldsymbol{replaceVar}(p, Skolem(x, l))) \\ \boldsymbol{skolem}(ForAll(x, p), l) & \twoheadrightarrow ForAll(x, \boldsymbol{skolem}(p, Cons(Var(x), l))) \\ \boldsymbol{replaceVar}(Predicate(s, xs), skl) & \twoheadrightarrow Predicate(s, \boldsymbol{replaceVar}(xs, skl)) \\ \boldsymbol{replaceVar}(And(p_1, p_2), skl) & \twoheadrightarrow And(\boldsymbol{replaceVar}(p_1, skl), \boldsymbol{replaceVar}(p_2, skl)) \\ \boldsymbol{replaceVar}(Or(p_1, p_2), skl) & \twoheadrightarrow Or(\boldsymbol{replaceVar}(p_1, skl), \boldsymbol{replaceVar}(p_2, skl)) \\ \boldsymbol{replaceVar}(Exists(x, p), skl) & \twoheadrightarrow Exists(x, \boldsymbol{replaceVar}(p, skl)) \\ \boldsymbol{replaceVar}(ForAll(x, p), skl) & \twoheadrightarrow ForAll(x, \boldsymbol{replaceVar}(p, skl)) \\ \boldsymbol{replaceVar}(Cons(Var(s), xs), Skolem(x, l)) & \twoheadrightarrow Cons(Skolem(x, l), \boldsymbol{replaceVar}(xs, Skolem(x, l))) \\ \boldsymbol{replaceVar}(Nil, x) & \twoheadrightarrow Nil \end{array}\right.$$

In Timbuk, the following rules are added to the CBTRS to encode and verify the desired properties expressed as annotations:

$$\left\{\begin{array}{ll} \boldsymbol{containExists}(Predicate(X, Y)) & \twoheadrightarrow false \\ \boldsymbol{containExists}(And(X, Y)) & \twoheadrightarrow \boldsymbol{bOr}(\boldsymbol{containExists}(X), \boldsymbol{containExists}(Y)) \\ \boldsymbol{containExists}(Or(X, Y)) & \twoheadrightarrow \boldsymbol{bOr}(\boldsymbol{containExists}(X), \boldsymbol{containExists}(Y)) \\ \boldsymbol{containExists}(Exists(X, Y)) & \twoheadrightarrow true \\ \boldsymbol{containExists}(Forall(X, Y)) & \twoheadrightarrow \boldsymbol{containExists}(Y) \\ \boldsymbol{bOr}(false, false) & \twoheadrightarrow false \\ \boldsymbol{bOr}(X, true) & \twoheadrightarrow true \\ \boldsymbol{bOr}(true, X) & \twoheadrightarrow true \end{array}\right.$$

B Proofs

Proposition 3.1. *Let $t \in \mathcal{T}(\mathcal{F},\mathcal{X}), p \in \mathcal{T}_\perp(\mathcal{C},\mathcal{X})$, t is p-free iff $\forall v \in [\![t]\!], v$ is p-free.*

Proof. If $t \in \mathcal{T}(\mathcal{C})$, then $[\![t]\!] = \{t\}$, hence the relation.

If $t \in \mathcal{T}(\mathcal{C},\mathcal{X})$, t is p-free iff $\forall \sigma$ such that $\sigma(t) \in \mathcal{T}(\mathcal{C})$, $\sigma(t)$ is p-free. Thus, by definition of the generalized ground semantics, t is p-free iff $\forall v \in [\![t]\!], v$ is p-free.

Finally, for $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, we proceed by induction on the number n of defined symbols in t. If $n = 0$, then $t \in \mathcal{T}(\mathcal{C},\mathcal{X})$, thus t verify the property. Given $n \geq 0$, we suppose that $\forall u \in \mathcal{T}(\mathcal{F},\mathcal{X})$ with $k \leq n$ defined symbols, u is p-free iff $\forall v \in [\![u]\!], v$ is p-free. We now consider $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ with $n+1$ defined symbols. By definition, t is p-free iff, $\forall \omega \in \mathcal{P}os\,(t)$ such that $t(\omega) = f_s^{-q} \in \mathcal{D}$, $t\,[v]_\omega$ is p-free, $\forall v \in \mathcal{T}_s(\mathcal{C})$ q-free. Moreover, given $\omega \in \mathcal{P}os\,(t)$ such that $t(\omega) = f_s^{-q} \in \mathcal{D}$ and $v \in \mathcal{T}_s(\mathcal{C})$ q-free, the number of defined symbols in $t\,[v]_\omega$ is less or equal to n, thus the inductive property guarantees that $t\,[v]_\omega$ is p-free iff $\forall w \in [\![t\,[v]_\omega]\!]$ is p-free. Finally, by definition of the generalized ground semantics, $w \in [\![t]\!]$ iff $\exists\, \omega \in \mathcal{P}os\,(t)$ with $t(\omega) = f_s^{-q}$, and $\exists\, v \in \mathcal{T}_s(\mathcal{C})$ q-free such that $w \in [\![t\,[v]_\omega]\!]$. Hence, t is p-free iff $\forall w \in [\![t]\!], w$ is p-free.

Therefore, by induction, $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is p-free iff $\forall v \in [\![t]\!], v$ is p-free.

Proposition 3.2. $\forall t \in \mathcal{T}(\mathcal{F},\mathcal{X}), [\![t]\!] = [\![\tilde{t}]\!]$

Proof. The proof is immediate by induction on the form of t.

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

$$\forall t, v \in \mathcal{T}(\mathcal{F}),\ \textit{if } t \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}} v,\ \textit{then } [\![v]\!] \subseteq [\![t]\!].$$

Proof. For all constructor rewrite rules $l \twoheadrightarrow r$, l is of the form $f(l_1,\ldots,l_n)$, thus we have $\forall \sigma, [\![\sigma(r)]\!] \subseteq [\![r]\!] \subseteq [\![l]\!] = [\![\sigma(f(l_1,\ldots,l_n))]\!]$. Moreover, $\forall t \in \mathcal{T}(\mathcal{F}), \forall u, v \in \mathcal{T}(\mathcal{C})$ $[\![v]\!] \subseteq [\![u]\!]$ implies $\forall \omega \in \mathcal{P}os\,(t)\,, [\![t\,[v]_\omega]\!] \subseteq [\![t\,[u]_\omega]\!]$. Therefore, by definition of the rewriting relation induced by such a semantics preserving rule, we have $\forall u, v \in \mathcal{T}(\mathcal{F})$, $u \longrightarrow_{\mathcal{R}} v$ implies $[\![v]\!] \subseteq [\![u]\!]$.

Proposition 4.1 (Pattern-free vs Deep Semantics). *Let $p \in \mathcal{T}(\mathcal{C},\mathcal{X}), t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, t is p-free iff $\{\!\{\tilde{t}\}\!\} \cap [\![p]\!] = \emptyset$.*

Proof. By definition, $\{\!\{\tilde{t}\}\!\} = \{u_{|\omega} \mid u \in [\![\tilde{t}]\!], \omega \in \mathcal{P}os\,(u)\} = \{u_{|\omega} \mid u \in [\![t]\!], \omega \in \mathcal{P}os\,(u)\}$, thus $\{\!\{\tilde{t}\}\!\} \cap [\![p]\!] = \emptyset$ if and only if $\forall u \in [\![t]\!], \omega \in \mathcal{P}os\,(u)\,, p \not\twoheadrightarrow u_{|\omega}$, *i.e.* $\forall u \in [\![t]\!], u$ is p-free. Therefore, thanks to Proposition 3.1, t is p-free if and only if $\{\!\{\tilde{t}\}\!\} \cap [\![p]\!] = \emptyset$.

Proposition 4.2. *For any constructor symbol $c \in \mathcal{C}$ and extended patterns $t_1,\ldots,t_n$, such that $\mathcal{D}om\,(c) = s_1 \times \cdots \times s_n$ and $t_1 : s_1, \ldots, t_n : s_n$, we have:*

- *If $\forall i \in [1,n], [\![t_i]\!] \neq \emptyset$, then $\{\!\{c(t_1,\ldots,t_n)\}\!\} = [\![c(t_1,\ldots,t_n)]\!] \cup \Big(\bigcup_{i=1}^{n} \{\!\{t_i\}\!\}\Big)$;*
- *If $\exists\, i \in [1,n], [\![t_i]\!] = \emptyset$, then $\{\!\{c(t_1,\ldots,t_n)\}\!\} = \emptyset$.*

Proof. If $\exists\, i \in [1,n], [\![t_i]\!] = \emptyset$, then $[\![c(t_1,\ldots,t_n)]\!] = \emptyset$. Hence, $\{\!\{c(t_1,\ldots,t_n)\}\!\} = \emptyset$.

Otherwise, $\forall i \in [1,n], [\![t_i]\!] \neq \emptyset$, and we consider the inclusions in the 2 directions separately. If $t \in \{\!\{c(t_1,\ldots,t_n)\}\!\}$, then $\exists\, u \in [\![c(t_1,\ldots,t_n)]\!], \omega \in \mathcal{P}os\,(u)$ such that $t = u_{|\omega}$. If $\omega = \epsilon$, then $t \in [\![c(t_1,\ldots,t_n)]\!]$, otherwise $\omega = i.\omega'$ with $i \in [1,n]$ and therefore $\exists\, u' \in [\![t_i]\!]$ such that $t = u'_{|\omega'}$, *i.e.* $t \in \{\!\{t_i\}\!\}$. Hence the direct inclusion. For the indirect inclusion, we can first remark that any $v \in [\![c(t_1,\ldots,t_n)]\!]$ is also in $\{\!\{c(t_1,\ldots,t_n)\}\!\}$. Let's now consider $v \in \{\!\{t_i\}\!\}$ for some i, *i.e.* $\exists\, u_i \in [\![t_i]\!], \omega \in \mathcal{P}os\,(u_i)$ such that $v = u_{i|\omega}$, then for all $u = c(u_1,\ldots,u_n)$ such that $\forall j \neq i, u_j \in [\![t_j]\!]$, $v = u_{|i.\omega}$, *i.e.* $v \in \{\!\{c(t_1,\ldots,t_n)\}\!\}$. Hence the indirect inclusion.

In order to prove Proposition 4.3, we introduce some auxiliary notions and prove some intermediate results.

First we analyse the deep semantics $\{\!\{x_s^{-p} \setminus r\}\!\}$. We can observe that $[\![x_s^{-p} \setminus r]\!] \subseteq [\![x_s \setminus (r+p)]\!]$, therefore if the latter is empty so is the first. Otherwise, we know that none of the patterns in $(r+p)$ is x_s. Moreover, for any $c, c' \in \mathcal{C}_s$, we have $[\![c(p_1,\ldots,p_n) \setminus c(q_1,\ldots,q_n)]\!] = [\![\sum_{i=1}^{n} c(p_1,\ldots,p_i \setminus q_i,\ldots,p_n)]\!]$ and $[\![c(p_1,\ldots,p_n) \setminus c'(q_1,\ldots,q_m)]\!] = [\![c(p_1,\ldots,p_n)]\!]$. Therefore, given c, r and p we can construct a set denoted $Q_c(r+p)$ of n-tuples $q = (q_1,\ldots,q_n)$, with n the arity of c, by successively distributing the patterns of $r+p$ that have the given constructor c as head (denoted $r^1,\ldots,r^k$), as follows:

$$
\begin{aligned}
[\![c(x_{s_1}^{-p},\ldots,x_{s_n}^{-p}) \setminus (r+p)]\!] &= [\![c(x_{s_1}^{-p},\ldots,x_{s_n}^{-p}) \setminus (r^1+\ldots+r^k)]\!] \\
&= [\![(c(x_{s_1}^{-p} \setminus \bot,\ldots,x_{s_n}^{-p} \setminus \bot) \setminus c(r_1^1,\ldots,r_n^1)) \setminus (r^2+\ldots+r^k)]\!] \\
&= [\![\left(\sum_{i\in[1,n]} c(x_{s_1}^{-p} \setminus \bot,\ldots,x_{s_i}^{-p} \setminus r_i^1,\ldots,x_{s_n}^{-p} \setminus \bot)\right) \setminus (c(r_1^2,\ldots,r_n^2)+\ldots+r^k)]\!] \\
&= [\![\left(\sum_{j\in[1,n]} \sum_{i\in[1,n]} c(x_{s_1}^{-p} \setminus \bot,\ldots,x_{s_j}^{-p} \setminus r_j^2,\ldots,x_{s_i}^{-p} \setminus r_i^1,\ldots,x_{s_n}^{-p} \setminus \bot)\right) \setminus (r^3+\ldots+r^k)]\!] \\
&\quad\vdots \\
&= [\![\sum_{q\in Q_c(r+p)} c(x_{s_1}^{-p} \setminus q_1,\ldots,x_{s_n}^{-p} \setminus q_n)]\!]
\end{aligned}
$$

Thus as shown in development (3), we get:

$$\{\!\{x_s^{-p} \setminus r\}\!\} = \begin{cases} \emptyset \text{ if } [\![x_s \setminus (r+p)]\!] = \emptyset \vee \forall c \in \mathcal{C}_s, Q'_c(r+p) = \emptyset \\ [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c\in\mathcal{C}_s} \bigcup_{q\in Q'_c(r+p)} \bigcup_{i\in[1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\} \text{ else} \end{cases}$$

with $Q'_c(u) = \{q \mid q \in Q_c(u) \wedge \forall i \in [1,n], [\![x_{s_i}^{-p} \setminus q_i]\!] \neq \emptyset\}$.

Given $Q_c(u)$ and $Q'c(u)$ defined above, we can now introduce the following abstractions for the deep semantics:

Definition B.1. *Let $p \in \mathcal{T}_\perp(\mathcal{C},\mathcal{X})$, a couple (s,r), with $s \in \mathcal{S}$ and r a sum of constructor patterns such that $r = \bot \vee r : s$, and S a finite set of such couples,*

- $\lfloor x_s^{-p} \setminus r \rfloor = \begin{cases} \emptyset \text{ if } [\![x_s \setminus (r+p)]\!] = \emptyset \vee \forall c \in \mathcal{C}_s, Q'_c(r+p) = \emptyset \\ \{(s,r)\} \cup \bigcup_{c\in\mathcal{C}_s} \bigcup_{q\in Q'_c(r+p)} \bigcup_{i\in[1,n]} \lfloor x_{s_i}^{-p} \setminus q_i \rfloor \text{ else} \end{cases}$

Such that, as shown above, we have $\{\!\{x_s^{-p} \setminus r\}\!\} = \bigcup_{(s',p') \in \lfloor x_s^{-p} \setminus r \rfloor} [\![x_{s'}^{-p} \setminus p']\!]$ *and* $\lfloor x_s^{-p} \setminus r \rfloor = \emptyset$ *iff* $[\![x_s^{-p} \setminus r]\!] = \emptyset$. *Thus* $Q'_c(u) = \{q \mid q \in Q_c(u) \wedge \forall i \in [1,n], \lfloor x_{s_i}^{-p} \setminus q_i \rfloor \neq \emptyset\}$.

– $\lfloor x_s^{-p} \setminus r \rfloor_S = \begin{cases} \emptyset \quad \textit{if} \quad [\![x_s \setminus (r+p)]\!] = \emptyset \\ S \textit{ else if } \exists (s,r') \in S, [\![r']\!] = [\![r]\!] \\ \emptyset \textit{ else if } \forall c \in \mathcal{C}_s, Q_c^{S \cup \{(s,r)\}}(r+p) = \emptyset \\ \{(s,r)\} \cup \bigcup\limits_{c \in \mathcal{C}_s} \bigcup\limits_{q \in Q_c^{S \cup \{(s,r)\}}(r+p)} \bigcup\limits_{i \in [1,n]} \lfloor x_{s_i}^{-p} \setminus q_i \rfloor_{S \cup \{(s,r)\}} \textit{ else} \end{cases}$

with $Q_c^S(u) = \{q \mid q \in Q_c(u) \wedge \forall i \in [1,n], \lfloor x_s^{-p} \setminus r \rfloor_S \neq \emptyset\}$

Moreover, we note $\{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus S} = \bigcup\limits_{(s',p') \in (\lfloor x_s^{-p} \setminus r \rfloor_S \setminus S)} [\![x_{s'}^{-p} \setminus p']\!]$

Lemma B.1. *Let* $p \in \mathcal{T}_\perp(\mathcal{C}, \mathcal{X})$, *a couple* (s,r), *with* $s \in \mathcal{S}$ *and* r *a sum of constructor pattern such that* $r = \perp$ *or* $r : s$, *and* S *a finite set of such couples. We have* $\lfloor x_s^{-p} \setminus r \rfloor \neq \emptyset$ *if and only if* $\lfloor x_s^{-p} \setminus r \rfloor_\emptyset \neq \emptyset$

Proof. We consider the following (possibly infinite) tree structure: $N = \langle (s',p'), c, q, L \rangle$ such that $c : s_1 \times \ldots \times s_n \mapsto s' \in \mathcal{C}, q \in Q_c(p'+p)$ and $L = [N_1, \ldots, N_n]$ with $\forall i, N_i$ is of the form $\langle (s_i, r_i), [\ldots] \rangle$ with $[\![r_i]\!] = [\![q_i]\!]$. We remark that $Q_c(p'+p)$ is correctly defined if and only $[\![x_{s'} \setminus (p'+p)]\!] \neq \emptyset$, then if such a tree exists, we have, for all nodes $\langle (s',p'), [\ldots] \rangle, [\![x_{s'} \setminus (p'+p)]\!] \neq \emptyset$.

As $[\![r]\!] = [\![r']\!]$ implies $[\![x_s^{-p} \setminus r]\!] = [\![x_s^{-p} \setminus r']\!]$, by construction, if $\{\!\{x_s^{-p} \setminus r\}\!\} \neq \emptyset$, then there exists such a tree. And conversely, if there exists such a tree, as for all nodes $\langle (s',p'), [\ldots] \rangle, [\![x_{s'} \setminus (p'+p)]\!] \neq \emptyset$, then we can construct a term t by assigning to each node the value of the constructor label, such that, by construction, $t : s$ and t is p-free, hence $t \in [\![x_s^{-p} \setminus r]\!]$, and thus $\{\!\{x_s^{-p} \setminus r\}\!\} \neq \emptyset$.

We prove now that $\lfloor x_s^{-p} \setminus r \rfloor_\emptyset \neq \emptyset$ if and only if there exists such a tree. If there exists such tree, we can prove that for each node $N = \langle (s',p'), [\ldots] \rangle$ of this tree $\lfloor x_{s'}^{-p} \setminus p' \rfloor_S \neq \emptyset$ with S the set of pairs (ζ, ρ) of each node in the path from the root of the tree to N. If the tree is finite, this is obviously true for each leaf. Otherwise, there is at least one infinite branch, and as each p' is a sum of a subterm r and subterms of p, there is only a finite number of such terms with a different ground semantics (as $[\![u+u]\!] = [\![u]\!]$). Hence, for each infinite branch, there is a node $N = \langle (\zeta, \rho), [\ldots] \rangle$ such that the path from the root of the tree to N contains a node $\langle (\zeta, \rho'), [\ldots] \rangle$ with $[\![\rho]\!] = [\![\rho']\!]$, hence $\lfloor x_{s'}^{-p} \setminus p' \rfloor_S$. We can then prove by induction that this holds for each node. Thus, we have, for the root node, $\lfloor x_s^{-p} \setminus r \rfloor_\emptyset \neq \emptyset$. If $\lfloor x_s^{-p} \setminus r \rfloor_\emptyset \neq \emptyset$, by construction of $\lfloor x_s^{-p} \setminus r \rfloor_\emptyset$, we can build a tree such that for each node $N = \langle (s',p'), [\ldots] \rangle$ of this tree $\lfloor x_{s'}^{-p} \setminus p' \rfloor_S \neq \emptyset$ with S the set of pairs (ζ, ρ) of each node in the path from the root of the tree to N, with each branch of the tree terminating on a node $\langle (\zeta, \rho), c, [\ldots] \rangle$ such that c is of arity 0 or there exists a node $\langle (\zeta, \rho'), [\ldots] \rangle$ with $[\![\rho]\!] = [\![\rho']\!]$ in which case we can repeat infinitely the path between the 2 nodes to get the desired tree.

Thus we have $\lfloor x_s^{-p} \setminus r \rfloor \neq \emptyset$ if and only if $\lfloor x_s^{-p} \setminus r \rfloor_\emptyset \neq \emptyset$.

Lemma B.2. *Let $p \in \mathcal{T}_\perp(\mathcal{C}, \mathcal{X})$, a couple (s, r), with $s \in \mathcal{S}$ and r a sum of constructor patterns such that $r = \perp \vee r : s$, and S a finite set of such couples. We have:*

1. *If $\lfloor x_s^{-p} \setminus r \rfloor_S \neq \emptyset$ then $\forall (s', p')$ with $q : s$ or $q = \perp$, $\lfloor x_s^{-p} \setminus r \rfloor_{S \cup \{(s',p')\}} \neq \emptyset$;*
2. *$\forall (s', p')$ with $p : s'$ or $p = \perp$, if $\lfloor x_s^{-p} \setminus r \rfloor_{S \cup \{(s',p')\}} \neq \emptyset$ then $\forall S'$ such that $\lfloor x_{s'}^{-p} \setminus p' \rfloor_{S'} \neq \emptyset$, $\lfloor x_s^{-p} \setminus r \rfloor_{S \cup S'} \neq \emptyset$;*
3. *$\forall (s', p')$, with $p : s'$ or $p = \perp$, if $\lfloor x_{s'}^{-p} \setminus p' \rfloor_S \neq \emptyset$, then $[\![x_s^{-p} \setminus r]\!]_{\setminus S \cup \{(s',p')\}} \subseteq [\![x_s^{-p} \setminus r]\!]_{\setminus S}$;*
4. *$\forall (s', p')$, with $p : s'$ or $p = \perp$, if $\lfloor x_{s'}^{-p} \setminus p' \rfloor_S \neq \emptyset$, then $[\![x_s^{-p} \setminus r]\!]_{\setminus S} \subseteq [\![x_s^{-p} \setminus r]\!]_{\setminus S \cup \{(s',p')\}} \cup [\![x_{s'}^{-p} \setminus p']\!]_{\setminus S}$.*

Proof. We consider $P(u)$ the set of all patterns that we can construct by sum of subterms of u and p, and $P_{\setminus S}(u) = \{t \mid t \in P(u) \wedge \forall (s', p') \in S, [\![t]\!] \neq [\![p']\!]\}$. Thanks to this, we can prove the 4 properties by induction on S and r, such that $P_{\setminus S}(r)$ is strictly decreasing.

The base case is for S and r such that $\exists (s, r') \in S$ with $[\![r]\!] = [\![r']\!]$, in this case $\lfloor x_s^{-p} \setminus r \rfloor_S = S$, and thus all 4 properties hold. Let now S be a set such that $\forall (s', r') \in S, s \neq s' \vee [\![r]\!] \neq [\![r']\!]$, we suppose that $\forall c \in \mathcal{C}_s, \forall q \in Q_c(r + p)$, the properties hold for all s_i, q_i and $S \cup \{(s, r)\}$, and we want to prove that they hold for S and r. Indeed, as q_i is a sum of subterms of r and p, we have $P(q_i) \subseteq P(r)$, and we have $r \in P_{\setminus S}(r)$ but $r \notin P_{\setminus S \cup \{(s,r)\}}(q_i)$, hence $P_{\setminus S \cup \{(s,r)\}}(q_i) \subset P_{\setminus S}(r)$.

1. If $\lfloor x_s^{-p} \setminus r \rfloor_S \neq \emptyset$ then $\forall (s', p')$ with $q : s$ or $q = \perp$, $\lfloor x_s^{-p} \setminus r \rfloor_{S \cup \{(s',p')\}} \neq \emptyset$;
   If $s' = s$ and $[\![p']\!] = [\![r]\!]$, then the property is obviously true. Otherwise, as $\lfloor x_s^{-p} \setminus r \rfloor_S \neq \emptyset$, we know that $[\![x_s \setminus (r + p)]\!] \neq \emptyset$ and that $\exists c \in \mathcal{C}_s$ such that $Q_c^{S \cup \{(s,r)\}}(r+p) \neq \emptyset$, *i.e.* $\exists q \in Q_c(r+p)$ such that $\forall i \in [1, n], \lfloor x_{s_i}^{-p} \setminus q_i \rfloor_{S \cup \{(s,r)\}} \neq \emptyset$. We can then apply the inductive property for $S \cup \{(s, r)\}$, hence $\forall i \in [1, n], \lfloor x_{s_i}^{-p} \setminus q_i \rfloor_{S \cup \{(s,r) \cup \{(s',p')\}} \neq \emptyset$, and thus $\lfloor x_s^{-p} \setminus r \rfloor_{S \cup \{(s',p')\}} \neq \emptyset$.
2. $\forall (s', p')$ with $p : s'$ or $p = \perp$, if $\lfloor x_s^{-p} \setminus r \rfloor_{S \cup \{(s',p')\}} \neq \emptyset$ then $\forall S'$ such that $\lfloor x_{s'}^{-p} \setminus p' \rfloor_{S'} \neq \emptyset$, $\lfloor x_s^{-p} \setminus r \rfloor_{S \cup S'} \neq \emptyset$;
   We proceed the exact same way.
3. $\forall (s', p')$, with $p : s'$ or $p = \perp$, if $\lfloor x_{s'}^{-p} \setminus p' \rfloor_S \neq \emptyset$, then $[\![x_s^{-p} \setminus r]\!]_{\setminus S \cup \{(s',p')\}} \subseteq [\![x_s^{-p} \setminus r]\!]_{\setminus S}$;
   If $s' = s$ and $[\![p']\!] = [\![r]\!]$, then the property is obviously true. Otherwise, as $\lfloor x_{s'}^{-p} \setminus p' \rfloor_S \neq \emptyset$, we have, thanks to the previous 2 properties, $\forall (s'', p'')$, $\lfloor x_{s''}^{-p} \setminus p'' \rfloor_{S \cup \{(s,r)\}} \neq \emptyset$ if and only if $\lfloor x_{s''}^{-p} \setminus p'' \rfloor_{S \cup \{(s,r),(s',p')\}} \neq \emptyset$. Thus, for all $c \in \mathcal{C}_s, Q_c^{S \cup \{(s',p'),(s,r)\}}(r + p) = Q_c^{S \cup \{(s',p')\}}(r + p)$. Therefore, if one is empty, so is the other, and both semantics then verify the property. Finally, if neither $[\![x_s \setminus (r + p)]\!]$ nor all $Q_c^S(r + p)$ are empty, we have:

$$\begin{aligned}
[\![x_s^{-p} \setminus r]\!]_{\setminus S \cup \{(s',p')\}} &= [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S \cup \{(s,r),(s',p')\}}(r+p)} \bigcup_{i \in [1,n]} [\![x_{s_i}^{-p} \setminus q_i]\!]_{\setminus S \cup \{(s,r),(s',p')\}} \\
&= [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S \cup \{(s,r)\}}(r+p)} \bigcup_{i \in [1,n]} [\![x_{s_i}^{-p} \setminus q_i]\!]_{\setminus S \cup \{(s,r),(s',p')\}} \\
&\subseteq [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S \cup \{(s,r)\}}(r+p)} \bigcup_{i \in [1,n]} [\![x_{s_i}^{-p} \setminus q_i]\!]_{\setminus S \cup \{(s,r)\}} \quad \text{by induction}
\end{aligned}$$

   And so $[\![x_s^{-p} \setminus r]\!]_{\setminus S \cup \{(s',q)\}} \subseteq [\![x_s^{-p} \setminus r]\!]_{\setminus S}$.

4. $\forall (s', p')$, with $p : s'$ or $p = \bot$, if $\lfloor x_{s'}^{-p} \setminus p' \rfloor_S \neq \emptyset$, then $\{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus S} \subseteq \{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus S \cup \{(s',p')\}} \cup \{\!\{x_{s'}^{-p} \setminus p'\}\!\}_{\setminus S}$;
   We procede the exact same way.

Proposition 4.3 (Correctness). *Given $s \in \mathcal{S}, p \in \mathcal{T}_\bot(\mathcal{C}, \mathcal{X})$ and $r : s$ a sum of constructor patterns, getReachable$(s, p, \emptyset, r)$ terminates and if we have $R =$ getReachable$(s, p, \emptyset, r)$, then*

$$\{\!\{x_s^{-p} \setminus r\}\!\} = \bigcup_{(s',p') \in R} [\![x_{s'}^{-p} \setminus p']\!]$$

Moreover, we have $\{\!\{x_s^{-p} \setminus r\}\!\} = \emptyset$ iff $R = \emptyset$.

Proof. If $\lfloor x_s^{-p} \setminus r \rfloor_S \neq \emptyset$, thanks to Lemma B.2, we have:

$$\begin{aligned}
\{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus S} &= [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S \cup \{(s,r)\}}(r+p)} \bigcup_{i \in [1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus S \cup \{(s,r)\}} \\
&= [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S}(r+p)} \bigcup_{i \in [1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus S \cup \{(s,r)\}} \\
&\subseteq [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S}(r+p)} \bigcup_{i \in [1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus S}
\end{aligned}$$

and:

$$\begin{aligned}
\{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus S} &= [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S \cup \{(s,r)\}}(r+p)} \bigcup_{i \in [1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus S \cup \{(s,r)\}} \\
&= [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S}(r+p)} \bigcup_{i \in [1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus S \cup \{(s,r)\}} \\
&= [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S}(r+p)} \bigcup_{i \in [1,n]} (\{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus S \cup \{(s,r)\}} \cup \{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus S}) \\
&\supseteq [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S}(r+p)} \bigcup_{i \in [1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus S}
\end{aligned}$$

Therefore, we then have:

$$\{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus S} = [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q_c^{S}(r+p)} \bigcup_{i \in [1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus S}$$

And thus, for $S = \emptyset$ and since $\lfloor x_s^{-p} \setminus r \rfloor_\emptyset = \emptyset \iff [\![x_s^{-p} \setminus r]\!] = \emptyset$:

$$\{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus \emptyset} = \begin{cases} \emptyset \text{ if } [\![x_s \setminus (r+p)]\!] = \emptyset \vee \forall c \in \mathcal{C}_s, Q'_c(r+p) = \emptyset \\ [\![x_s^{-p} \setminus r]\!] \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in Q'_c(r+p)} \bigcup_{i \in [1,n]} \{\!\{x_{s_i}^{-p} \setminus q_i\}\!\}_{\setminus \emptyset} \text{ else} \end{cases}$$

This relation is equivalent to the one for $\{\!\{x_s^{-p} \setminus r\}\!\}$, hence: $\{\!\{x_s^{-p} \setminus r\}\!\}_{\setminus \emptyset} = \{\!\{x_s^{-p} \setminus r\}\!\}$

Finally, by looking at the algorithm, we can observe that $getReachable(s, p, S, r) = \lfloor x_s^{-p} \setminus r \rfloor_S$. To do so, we reference each **return** case of the algorithm by $(R1), (R2), (R3)$ and $(R4)$, in order of appearance.

The algorithm starts by conflating r with $r + p$ when $p : s$ (otherwise, p has no effect), thanks to the first *if* of the algorithm. Thanks to the second *if* we then have an empty **return** on $(R1)$ when $[\![x_s \setminus r]\!] = \emptyset$. And the third *if* leads to **returning** S on $(R2)$ when $\exists\, (s, r') \in S, [\![r']\!] = [\![r]\!]$.

If the algorithm did not **return** on $(R1)$ or $(R2)$, we then have, with the conflated r, $[\![x_s^{-p} \setminus r]\!] = [\![\sum_{c \in \mathcal{C}_s} c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus r]\!]$. Thus the algorithm loops on

$c \in \mathcal{C}_s$. The first nested *for* loop computes the set $Q_c(r)$ obtained, as mentioned, by successively distributing the patterns of r that have the given constructor c as head. The second *for* loop then recursively calls `getReachable` on the couples (s_i, q_i) obtained this way, and updates R with the results obtained.

Moreover, by considering $P_{\setminus A}(u) = \{t \mid t \in P(u) \wedge \forall (s', p') \in A, [\![t]\!] \neq [\![p']\!]\}$, with $P(u)$ the (finite) set of all patterns that we can construct by sum of subterms of u and p, we can remark that $\forall c \in \mathcal{C}_s, q \in Q_c(r), i < arity\,(c)$ we have $P_{\setminus (S \cup \{(s,r)\})}(q_i) \subsetneq P_{\setminus S}(r)$, thus guaranteeing the termination of all recursion chains. Indeed, as q_i is a sum of subterms of r and p, we have $P(q_i) \subseteq P(r)$, and, as the algorithm did not return on $(R2)$, we have $r \in P_{\setminus S}(r)$ but $r \notin P_{\setminus S \cup \{(s,r)\}}(q_i)$, hence $P_{\setminus S \cup \{(s,r)\}}(q_i) \subsetneq P_{\setminus S}(r)$.

Finally, as the algorithm did not **return** on $(R1)$ we have $[\![x_s^{-p} \setminus r]\!] = \emptyset$ if and only if, $\forall c \in \mathcal{C}_s, Q'_c(r) = \emptyset$, hence the boolean variable *reachable* that stays false when $\forall c \in \mathcal{C}_s, Q_c^{S \cup \{(s,r)\}}(r) = \emptyset$, resulting in an empty **return** $(R4)$. Similarly, $[\![c(x_{s_1}^{-p} \setminus q_1, \ldots, x_{s_n}^{-p} \setminus q_n)]\!]$ is empty if and only if $\exists\, i$ such that $[\![x_{s_i}^{-p} \setminus q_i]\!] = \emptyset$, so R is updated with the result of the recursive calls for a given $c \in \mathcal{C}_s$ and $q \in Q_c(r)$ only if none of these recursive calls returns an empty result. We thus have the concatenated result as described in the definition of $\lfloor x_s^{-p} \setminus r \rfloor_S$ on **return** $(R3)$.

Proposition 4.4 (Semantics preservation). *For any extended patterns p, q, if $p \twoheadrightarrow_{\mathcal{R}_p} q$ then $[\![p]\!] = [\![q]\!]$.*

Proof. We prove that the ground semantics of the left-hand side and right-hand side of the rewrite rules of $\mathcal{R}_p$ are the same.

In the case of the rule (E1), as we have $[\![\delta(p_1, \ldots, p_n)]\!] = \{\delta(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in [\![p_1]\!] \times \ldots \times [\![p_n]\!]\}$ and the ground semantics of $\bot$ is empty, so is the semantics of $\delta(v_1, \ldots, \bot, \ldots, v_n)$. Hence the equality of ground semantics of the 2 sides of the rule. For the rule (S1), we have:

$$\begin{aligned}
[\![\delta(v_1, \ldots, v_i + w_i, \ldots, v_n)]\!] &= \{\delta(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in [\![v_1]\!] \times \ldots \times [\![v_i + w_i]\!] \times \ldots \times [\![v_n]\!]\} \\
&= \{\delta(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in [\![v_1]\!] \times \ldots \times [\![v_i]\!] \cup [\![w_i]\!] \times \ldots \times [\![v_n]\!]\} \\
&= \{\delta(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in [\![v_1]\!] \times \ldots \times [\![v_i]\!] \times \ldots \times [\![v_n]\!]\} \\
&\quad \cup \{\delta(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in [\![v_1]\!] \times \ldots \times [\![w_i]\!] \times \ldots \times [\![v_n]\!]\} \\
&= [\![\delta(v_1, \ldots, v_i, \ldots, v_n)]\!] \cup [\![\delta(v_1, \ldots, w_i, \ldots, v_n)]\!]
\end{aligned}$$

For rules (M7) and (T3), we consider both inclusions separately:

(M7): If $v \in [\![\alpha(v_1, \ldots, v_n) \setminus \alpha(t_1, \ldots, t_n)]\!]$, then $v \in [\![\alpha(v_1, \ldots, v_n)]\!]$ and $v \notin [\![\alpha(t_1, \ldots, t_n)]\!]$. As $[\![\alpha(p_1, \ldots, p_n)]\!] = \{\alpha(w_1, \ldots, w_n) \mid (w_1, \ldots, w_n) \in [\![p_1]\!] \times \ldots \times [\![p_n]\!]\}$, $v = \alpha(w_1, \ldots, w_n)$ such that $\forall i \in [1, n], w_i \in [\![v_i]\!]$ and $\exists\, j \notin [1, n], w_j \in [\![t_j]\!]$. Therefore, $w_j \in [\![v_j]\!] \setminus [\![t_j]\!]$ and thus $v \in [\![\alpha(v_1, \ldots, v_j \setminus t_j, \ldots, v_n)]\!]$, and finally $v \in [\![\sum_{k \in [1,n]} \alpha(v_1, \ldots, v_k \setminus t_k, \ldots, v_n)]\!]$. Hence the first inclusion. We can show that if $v \in [\![\sum_{k \in [1,n]} \alpha(v_1, \ldots, v_k \setminus t_k, \ldots, v_n)]\!]$, then $v \in [\![\alpha(v_1, \ldots, v_n) \setminus \alpha(t_1, \ldots, t_n)]\!]$ similarly in order to prove the second inclusion.

(T3): If $v \in [\![\alpha(v_1, \ldots, v_n) \times \alpha(w_1, \ldots, w_n)]\!]$, then $v \in [\![\alpha(v_1, \ldots, v_n)]\!]$ and $v \in [\![\alpha(w_1, \ldots, w_n)]\!]$. As $[\![\alpha(p_1, \ldots, p_n)]\!] = \{\alpha(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in [\![p_1]\!] \times \ldots \times [\![p_n]\!]\}$, $v = \alpha(t_1, \ldots, t_n)$ such that $\forall i \in [1, n], t_i \in [\![v_i]\!]$ and $t_i \in [\![w_i]\!]$.

Therefore, $\forall i \in [1,n], t_i \in [\![v_i]\!] \cap [\![w_i]\!]$ and thus $v \in [\![\alpha(v_1 \times w_1, \ldots, v_n \times w_n)]\!]$. Hence the first inclusion. Similarly, we can show that if $v \in [\![\alpha(v_1 \times w_1, \ldots, v_n \times w_n)]\!]$, then $v \in [\![\alpha(v_1, \ldots, v_n) \times \alpha(w_1, \ldots, w_n)]\!]$, to prove the second inclusion.

For the rest of the rules but (P1), the definition of ground semantics of extended patterns and properties of set operations give us the equality between the ground semantics of 2 side of each rule in a fairly straightforward manner. In particular, in the case of rules (M1), (T1) and (T2), we can remark that, as we only consider well-sorted extended patterns, in these 3 rules we have $v : s$ and therefore $[\![v]\!] \subseteq [\![x_s^{-\perp}]\!]$. Hence the equality of ground semantics of the 2 sides of these rules.

Finally, for (P1), we first prove that:

$$[\![x_s^{-p}]\!] = \bigcup_{c \in \mathcal{C}_s} [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus p]\!] \tag{1}$$

Let's consider both inclusion separately. Let $t \in \bigcup_{c \in \mathcal{C}_s} [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus p]\!]$, *i.e.* $\exists\, c \in \mathcal{C}_s$ such that $t \in [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p})]\!]$ and $t \notin [\![p]\!]$. Thus $\exists\, (t_1, \ldots, t_n) \in [\![x_{s_1}^{-p}]\!] \times \ldots [\![x_{s_n}^{-p}]\!]$ such that $p \not\preccurlyeq t = c(t_1, \ldots, t_n)$. Therefore, $t : s$ and $\forall \omega \in \mathcal{P}os\,(t)\,, p \not\preccurlyeq t_{|\omega}$, *i.e.* t is p-free. Hence $t \in [\![x_s^{-p}]\!]$. Let $t \in [\![x_s^{-p}]\!]$, then $t : s$ and t is p-free. Thus $\exists\, c : s_1 \times \ldots \times s_n \mapsto s, (t_1, \ldots, t_n) \in \mathcal{T}_{s_1}(\mathcal{C}) \times \ldots \times \mathcal{T}_{s_n}(\mathcal{C})$ such that $t = c(t_1, \ldots, t_n)$ with $\forall i \in [1,n], t_i$ is p-free. Therefore, $t \in [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p})]\!]$ and, as $p \not\preccurlyeq t$, $t \notin [\![p]\!]$. Hence $t \in [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus p]\!]$, and finally $t \in \bigcup_{c \in \mathcal{C}_s} [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus p]\!]$.

Therefore, we have $[\![x_s^{-p}]\!] = [\![\sum_{c \in \mathcal{C}_s} c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus p]\!]$ hence

$$\begin{aligned}
[\![x_s^{-p} \times \alpha(v_1, \ldots, v_n)]\!] &= [\![\big(\textstyle\sum_{c \in \mathcal{C}_s} c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus p\big) \times \alpha(v_1, \ldots, v_n)]\!] \\
&= \left(\bigcup_{c \in \mathcal{C}_s} [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \setminus p]\!] \right) \cap [\![\alpha(v_1, \ldots, v_n)]\!] \\
&= \left(\bigcup_{c \in \mathcal{C}_s} [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p})]\!] \setminus [\![p]\!] \right) \cap [\![\alpha(v_1, \ldots, v_n)]\!] \\
&= \bigcup_{c \in \mathcal{C}_s} [\![c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p})]\!] \cap \big([\![\alpha(v_1, \ldots, v_n)]\!] \setminus [\![p]\!]\big) \\
&= [\![\sum_{c \in \mathcal{C}_s} c(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \times \big(\alpha(v_1, \ldots, v_n) \setminus p\big)]\!]
\end{aligned}$$

Lemma B.3 (Convergence). *The rewriting system $\mathcal{R}_p$ is confluent and terminating.*

Proof. A meta-encoding of a complete approximation of the rule schema $\mathcal{R}_p$ is provided in Appendix C. Automatic termination proof tools such as TTT2 and AProVE have been used to prove that this meta-encoding is terminating and we can thus directly conclude to the termination of $\mathcal{R}_p$

We show the local confluence of the system by proving that all critical pairs induced by rewrite rules of the system converge. We have the following critical pairs:
$(A1) - (A2)$ (converge directly),

$(A1) - (S1)$ and $(A2) - (S1)$ (converge with $E1$ and $A1/A2$),
$(A1) - (S2)$ and $(A2) - (S2)$ (converge with $E2$ and $A1/A2$),
$(A1) - (S3)$ and $(A2) - (S3)$ (converge with $E3$ and $A1/A2$),
$(A1) - (M3)$ and $(A2) - (M3)$ (converge with $M5$ and $A1/A2$),
$(A1) - (M6)$ and $(A2) - (M3)$ (converge with $M2$),
$(E1) - (M6)$ (converge with $M5$ and $E1$, twice $M5$),
$(E1) - (M7)$ only left possible (converge with $M5$ and $M2$, n times $E1$, n times $A1/A2$),
$(E1) - (M8)$ left (converge with $M5$ and $E1$),
$(E1) - (M8)$ right (converge with $M2$),
$(E2) - (E3)$ (converge directly),
$(E2) - (S3)$ and $(E3) - (S2)$ (converge with twice $S2/S3$, $A1/A2$),
$(E2) - (T1)$ and $(E3) - (T2)$ (converge directly),
$(S1) - (M6)$ (converge with $M3$, twice $M6$ and $S1$, twice $M3$),
$(S1) - (M7)$ only left possible (converge with $M3$, twice $M7$ and $M3$, n times $S1$),
$(S1) - (M8)$ left (converge with $M3$, twice $M8$ and $S1$),
$(S1) - (M8)$ right (converge with $M6$, twice $M8$),
$(S1) - (T3)$ left (converge with $S2$, twice $T3$ and $S2$, $S1$),
$(S1) - (T3)$ right (converge with $S3$, twice $T3$ and $S3$, $S1$),
$(S1) - (T4)$ left (converge with $S2$, twice $T4$, $A1/A2$),
$(S1) - (T4)$ right (converge with $S3$, twice $T4$, $A1/A2$),
$(S1) - (P1)$ (converge with $S3$, twice $P1$ and $S1$, $M3$, $S3$),
$(S2) - (S3)$ (converge with $S3$ and $S2$),
$(S2) - (T1)$ (converge with twice $T1$),
$(S3) - (P4)$ (converge with twice $P4$ and $S3$, twice $M6$),
$(S3) - (T2)$ (converge with twice $T1$),
$(M1) - (M3)$ (converge with twice $M1$, $A1/A2$),
$(M1) - (M5)$ (converge directly),
$(M1) - (P6)$ (converge directly),
$(M2) - (M3)$ (converge with twice $M1$),
$(M2) - (M5)$ (converge directly),
$(T1) - (T2)$ (converge directly),
$(T1) - (P4)$ (converge with $T1$),
$(T2) - (P3)$ (converge with $T2$).

In order to prove Proposition 4.5, we first prove the conservation of the *regular* property with Lemma B.4 and we introduce the notion of *depth* of an extended pattern, in order to show Lemmas B.5, B.6 and B.7.

Lemma B.4. *Let u and v extended patterns such that $u \longrightarrow_{\mathcal{R}_p} v$, then*

$$u \text{ regular} \Longrightarrow v \text{ regular}$$

Proof. We can first remark that no rule in $\mathcal{R}_p$ modifies variable annotation, and the only rule that introduce new annotated variables, *i.e.* $(P1)$, annotates the new variables with a pattern annotation from the origin term, *i.e.* $\perp$ for

a *regular* term. Moreover, forall extended pattern t and u, $\forall \omega \in \mathcal{P}os\,(t)\,, t\,[u]_\omega$ *regular* implies u *regular* and forall *regular* extended v, $t\,[v]_\omega$ is *regular*. Thus all rules preserve the *regular* property.

Definition B.2 (Depth). *We define the notion of* depth *of an extended pattern:*

- $depth(x_s^{-p}) = depth(x_s^{-p} \setminus w) = depth(\bot) = 1$
- $depth(c(t_1, \ldots, t_n)) = 1 + max_{i \in [1,n]}(depth(t_i))$
- $depth(t_1 + t_2) = max(depth(t_1), depth(t_2))$
- $depth(t_1 \setminus t_2) = depth(t_1)$

Lemma B.5. *If t is a (quasi-)additive pattern, then $t \downarrow_{\mathcal{R}_p}$ is a (quasi-)additive pattern such that $depth(t \downarrow_{\mathcal{R}_p}) \leq depth(t)$.*

Moreover the normal form $t \downarrow_{\mathcal{R}_p} = v$ is either $\bot$ or a pure term such that $\forall \omega_+ \in \mathcal{P}os\,(v)\,, v(\omega_+) = + \implies \forall \omega < \omega_+, v(\omega) = +$ and $[\![t]\!] = \emptyset$ if and only if $v = \bot$.

Proof. We can first observe that the only rule that apply on an additive pattern are A1, A2, E1 and S1 (and the same rules plus M1 and P6 for quasi-additive patterns). Moreover, for each rule, it reduces a (quasi-)additive pattern into a (quasi-)additive pattern. Therefore, the normal form of an (quasi-)additive pattern, is indeed an (quasi-)additive pattern.

Moreover, the *depth* measure induces a monotonic ordering over quasi-additive patterns with regard to the $\leq$ operators, *i.e.* $depth(u) \leq depth(v)$ implies $depth(t\,[u]_\omega) \leq depth(t\,[v]_\omega)$. Finally, as the *depth* is decreasing on all applicable rules, we know that $depth(t \downarrow_{\mathcal{R}_p}) \leq depth(t)$.

Let's now suppose that $v = t \downarrow_{\mathcal{R}_p}$ contains a sum below a constructor, *i.e.* contains a subterm of the form $c(v_1, \ldots, v_i + u_i, \ldots, v_n)$, which would be a redex for S1, and thus v would not be a normal form. Therefore, v does not contain a sum below a constructor.

Finally, if $t \downarrow_{\mathcal{R}_p} = \bot$ then Proposition 4.4 ensures that $[\![t]\!] = [\![\bot]\!] = \emptyset$. We note $v = t \downarrow_{\mathcal{R}_p}$, once again we know that $[\![t]\!] = [\![v]\!]$, let's prove that if $[\![v]\!] = \emptyset$ then $v = \bot$. We suppose that $v \neq \bot$ and we prove by induction that v is not in normal form.

If $v = x_s^{-p} \setminus u$ and $[\![v]\!] = \emptyset$, then rule P6 applies, thus v is not in normal form. If $v = v_1 + v_2$, then $[\![v_1]\!] = [\![v_2]\!] = \emptyset$, thus by induction, either $v_1 = v_2 = \bot$ in which case both A1 and A2 applies, or at least one of them is not in normal form. In both cases, v is not in normal form. Finally, if $v = c(v_1, \ldots, v_n)$, then $\exists\, i \in [1, n]$ such that $[\![v_i]\!] = \emptyset$, thus by induction, either $v_i = \bot$ and rule E1 applies or v_i is not in normal form. In both cases, v is not in normal form.

Therefore, if $v \neq \bot$ is a quasi-additive such that $[\![v]\!] = \emptyset$, then v is not in normal form. Thus, if $[\![t]\!] = \emptyset$, then $t \downarrow_{\mathcal{R}_p} = \bot$.

Lemma B.6. *If t a quasi-additive pattern and u a regular additive pattern, then $v = (t \setminus u) \downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern such that $depth(v) \leq depth(t \setminus u)$.*

Proof. We can remark that $\mathcal{R}_p$ preserves the *regular* property of a pattern, thus, thanks to Lemma B.5, we can suppose, by confluence, that t and u are in normal form, we then prove this lemma by induction on the form of t and u.

In the case when $u = x_s^{-\perp}$, rule M1 applies to $t \setminus u$ and reduces it to $v = \perp$ which cannot be reduced furthermore. Moreover $depth(\perp) = 1 \leq depth(t \setminus u)$. In the case when $u = \perp$, rule M2 applies to $t \setminus u$ and reduces it to t, and as we supposed t in normal form, it cannot be reduced anymore. Moreover $depth(t) = depth(t \setminus u)$.

In other cases, we proceed by induction:

- if $u = u_1 + u_2$, we have to consider the different form of t. If $t = x_s^{-p}$, the only rule that can apply is P6, which, if it does, reduces $t \setminus u$ to $\perp$, thus, either way, the term v obtained cannot be reduced furthermore and is quasi-additive pattern with $depth(v) = 1 = depth(t \setminus u)$. If $t = x_s^{-p} \setminus u'$, the only rules that can apply are P5 and P6. If P5 applies, it reduces $t \setminus u$ to $x_s^{-p} \setminus (u' + u)$ for which only P6 can apply. If P6 applies, it reduces $t \setminus u$ to $\perp \setminus u$, which is then reduced to $\perp$ by M5, and $x_s^{-p} \setminus (u' + u)$ to $\perp$. In all cases, the term v obtained cannot be reduced furthermore and is quasi-additive pattern with $depth(v) = 1 = depth(t \setminus u)$. If $t = c(t_1, \ldots, t_n)$, rule M6 applies to $t \setminus u$ and reduces it to $(t \setminus u_1) \setminus u_2$. Moreover, by induction on u, $v' = (t \setminus u_1) \downarrow_{\mathcal{R}_p}$ and $v = (v' \setminus u_2) \downarrow_{\mathcal{R}_p}$ are both quasi-additive patterns such that $depth(v) \leq depth(v') \leq depth(t \setminus u)$. Hence, by confluence, $v = (t \setminus u) \downarrow_{\mathcal{R}_p}$. Finally, if $t = t_1 + t_2$, then rule M3 applies to $t \setminus u$ and reduces it to $(t_1 \setminus u) + (t_2 \setminus u)$. Moreover, by induction on t, $v_1 = (t_1 \setminus u) \downarrow_{\mathcal{R}_p}$ and $v_2 = (t_2 \setminus u) \downarrow_{\mathcal{R}_p}$ are both quasi-additive patterns such that $depth(v_1) \leq depth(t \setminus u)$ and $depth(v_2) \leq depth(t \setminus u)$. Therefore, $depth(v_1 + v_2) \leq depth(t \setminus) u$, and $t \setminus u \twoheadrightarrow_{\mathcal{R}_p} v_1 + v_2$. So as $v_1 + v_2$ is a quasi-additive pattern we know, thanks to Lemma B.5, that $v = (v_1 + v_2) \downarrow_{\mathcal{R}_p} = (t \setminus u)_{\mathcal{R}_p}$ is a quasi-additive pattern such that $depth(v) \leq depth(v_1 + v_2) \leq depth(t \setminus u)$.
- if $u = c(u_1, \ldots, u_n)$, with $\forall i \in [1, n], u_i$ a regular symbolic pattern (as u is in normal form), we have to consider the different form of t. If $t = x_s^{-p}$, the only rule that can apply is P6, which, if it does, reduces $t \setminus u$ to $\perp$, thus, either way, the term v obtained cannot be reduced furthermore and is quasi-additive pattern with $depth(v) = 1 = depth(t \setminus u)$. If $t = x_s^{-p} \setminus u'$, the only rules that can apply are P5 and P6. If P5 applies, it reduces $t \setminus u$ to $x_s^{-p} \setminus (u' + u)$ for which only P6 can apply. If P6 applies, it reduces $t \setminus u$ to $\perp \setminus u$, which is then reduced to $\perp$ by M5, and $x_s^{-p} \setminus (u' + u)$ to $\perp$. In all cases, the term v obtained cannot be reduced furthermore and is quasi-additive pattern with $depth(v) = 1 = depth(t \setminus u)$. For the case when $t = c'(t_1, \ldots, t_m)$, if $c \neq c'$, rule M8 applies to $t \setminus u$ and reduces it to t. Otherwise rule M7 applies to $t \setminus u$ and reduces it to $\sum_{i \in [1,n]} c(t_1, \ldots, t_i \setminus u_i, \ldots, t_n)$, and by induction on t, $\forall i, v_i = (t_i \setminus u_i) \downarrow_{\mathcal{R}_p}$ a quasi-additive pattern such that $depth(v_i) \leq depth(t_i \setminus u_i)$. Therefore, $t \setminus u \twoheadrightarrow_{\mathcal{R}_p} \sum_{i \in [1,n]} c(t_1, \ldots, v_i, \ldots, t_n)$ and by monotonicity, $depth(\sum_{i \in [1,n]} c(t_1, \ldots, v_i, \ldots, t_n)) \leq depth(t \setminus u)$. Moreover, $w = \sum_{i \in [1,n]} c(t_1, \ldots, v_i, \ldots, t_n)$ is a quasi-additive pattern, so according to Lemma B.5, $v = w \downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern such that $depth(v) \leq depth(w)$. Hence, by conflu-

ence, $v = (t \setminus u)\downarrow_{\mathcal{R}_p}$ and $depth(v) \leq depth(t \setminus u)$. Finally, if $t = t_1 + t_2$, we proceed identically as for $u = u_1 + u_2$.

Lemma B.7. *If t a quasi-additive pattern and u a regular quasi-additive pattern, then $(t \times u)\downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern.*

Proof. We can remark that $\mathcal{R}_p$ preserves the *regular* property of a pattern, thus, thanks to Lemma B.5, we can suppose, by confluence, that t and u are in normal form, we then prove this lemma by induction on the depth of u and the form of t and u.

The base case is for u such that $depth(u) = 1$. We proceed by induction on the form u such that $depth(u) = 1$. If $u = x_s^{-\perp}$, the only rule that applies to $t \setminus u$ is T2, which reduces it to t.

If $u = x_s^{-\perp} \setminus v$ with v a regular additive pattern, we proceed by induction on t. If $t = c(t_1, \ldots, t_n)$ or $t = x_s^{-p}$ or $t = x_s^{-p} \setminus w$ the only rules that applies to $t \setminus u$ are, respectively, P2 or P3 or P4, which with T1 reduces it to $t \setminus v$. And thanks to Lemma B.6 we know that $(t \setminus v)\downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern. Finally, if $t = t_1 + t_2$, then rule S2 applies to $t \setminus u$ and reduces it to $(t_1 \times u) + (t_2 \times u)$. Moreover, by induction on t_1 and t_2, $v_1 = (t_1 \times u)\downarrow_{\mathcal{R}_p}$ and $v_2 = (t_2 \times u)\downarrow_{\mathcal{R}_p}$ are both quasi-additive patterns. So, as $v_1 + v_2$ is a quasi-additive pattern, we know, thanks to Lemma B.5, that $(v_1 + v_2)\downarrow_{\mathcal{R}_p} = (t \setminus u)\downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern.

If $u = c()$, we proceed by induction on the form of t. If $t = c()$, then rule T3 applies to $t \times u$ and reduces it to $c()$. If $t = c'(t_1, \ldots, t_n)$ with $c' \neq c$, then rule T4 applies to $t \setminus u$ and reduces it to $\perp$. If $t = x_s^{-p}$, then rule P1 applies to $t \times u$ and reduces it to $\sum_{d \in \mathcal{C}_s} d(x_{s_1}^{-p}, \ldots, x_{s_n}^{-p}) \times (c() \setminus p)$. If $p = c()$, then rule M7 applies to $c() \setminus p$ and reduces it to $\perp$, leading to rule E2 applying and ultimately reducing $t \setminus u$ to $\perp$. If $p = c'(p_1, \ldots, p_n)$ with $c' \neq c$, then rule M8 applies to $c() \setminus p$ and reduces it to $c()$, and because we only consider well sorted extended patterns, $c \in \mathcal{C}_s$, thus repeatedly applying S3, T4, A1/ A2 and finally T3 ultimately reduces $t \setminus u$ to $c()$. Finally, if $t = t_1 + t_2$, we proceed exactly the same way as in the case when $u = x_s^{-\perp} \setminus v$ to prove the induction step on the form of t.

Finally, if $u = u_1 + u_2$ with $depth(u_1) = depth(u_2) = 1$, then rule S3 applies to $t \setminus u$ and reduces it to $(t \times u_1) + (t \times u_2)$. Moreover, by induction on $u1$ and u_2, $v_1 = (t \times u_1)\downarrow_{\mathcal{R}_p}$ and $v_2 = (t \times u_2)\downarrow_{\mathcal{R}_p}$ are both quasi-additive patterns. So as $v_1 + v_2$ is a quasi-additive pattern we know, thanks to Lemma B.5, that $(v_1 + v_2)\downarrow_{\mathcal{R}_p} = (t \setminus u)_{\mathcal{R}_p}$ is a quasi-additive pattern.

We now suppose $depth(u) = n > 1$ and for all quasi-additive pattern υ such that $depth(\upsilon) < n$, for all quasi-additive pattern τ, $(\tau \setminus \upsilon)\downarrow_{\mathcal{R}_p}$ is quasi-additive pattern. Let's prove by induction on the form of t and u that, for all quasi-additive pattern t, $(t \setminus u)\downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern.

If $u = c(u_1, \ldots, u_m)$ with $\forall i \in [1, m], depth(u_i) < n$, we proceed by induction on the form of t. For the case when $t = c'(t_1, \ldots, t_{m'})$, if $c \neq c$, then rule T4 applies to $t \setminus u$ and reduces it to $\perp$. Otherwise, rule T3 applies to $t \times u$ and reduces it to $c(t_1 \times u_1, \ldots, t_n \times u_m)$. Moreover, by induction on the depth of u

we know that $\forall i \in [1,m], v_i = (t_i \times u_i) \downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern, thus $c(v_1,\ldots,v_m)$ is a quasi-additive pattern and we know, thanks to Lemma B.5, that $c(v_1,\ldots,v_m) \downarrow_{\mathcal{R}_p} = (t \setminus u) \downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern. If $t = x_s^{-p}$, then rule T4 applies to $t \setminus u$ and reduces it to $\sum_{d \in \mathcal{C}_s} d(x_{s_1}^{-p},\ldots,x_{s_{m'}}^{-p}) \times (u \setminus p)$. Thanks to Lemma B.6, we know that $(u \setminus p) \downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern which depth is less than or equal to n, and we can easily show that its either $\bot$ or a sum of quasi-additive patterns of the form $c(w_1,\ldots,w_n)$, with $\forall i \in [1,m], \mathit{depth}(w_i) < n$. If $(u \setminus p) \downarrow_{\mathcal{R}_p} = \bot$, then rule E3 applies and thus $(t \times u) \downarrow \mathcal{R}_p = \bot$. Otherwise, by applying recursively S2/S3, T4/T3, and A1/A2 we get $t \times u \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}_p} \sum c(x_{s_1}^{-p} \times w_1,\ldots,x_{s_m}^{-p} \times w_m)$. Moreover, by induction on the depth of u we know that $\forall i \in [1,m], v_i = (t_i \times w_i) \downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern, thus $\sum c(v_1,\ldots,v_m)$ is a quasi-additive pattern and we know, thanks to Lemma B.5, that $(\sum c(v_1,\ldots,v_m)) \downarrow_{\mathcal{R}_p} = (t \setminus u) \downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern. Finally, if $t = t_1 + t_2$, we proceed exactly the same way as in the case when $u = x_s^{-\bot} \setminus v$ to prove the induction step on the form of t.

Finally, if $u = u_1 + u_2$, we proceed exactly the same way as in the case $\mathit{depth}(u) = 1$ to prove the induction step on the form u when $\mathit{depth}(u) > 1$.

Proposition 4.5. *The rewriting system $\mathcal{R}_p$ is confluent and terminating. Given a quasi-additive pattern t and a constructor pattern p, we have $t \times p \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}_p} \bot$ if and only if $[\![t \times p]\!] = \emptyset$.*

Proof. Confluence and termination are proved in Lemma B.3.

Based on Lemma B.7, we know that $(t \times p) \downarrow_{\mathcal{R}_p}$ is a quasi-additive pattern. Moreover, according to Proposition 4.4, the semantics of $t \times p$ is empty if and only if the semantics of its normal form is empty, hence, thanks to Lemma B.5, if and only if its normal form is $\bot$.

C Meta encoding of the rewriting system $\mathcal{R}_p$

The meta encoding of the rule schemas in Figure 2 is given below in a syntax usable by AProVE/TTT2. Both AProVE and TTT2 can be used to prove the termination of this rewriting system.

```
(VAR u u1 u2 v v1 v2 w f g lu lv l lacc n m i tail sig p q)
(RULES
  plus(bot,v) -> v
  plus(v,bot) -> v


  appl(f,lv) -> split(f,lv,nil)
    split(f,cons(u,lu),lv) -> split(f,lu,cons(u,lv))
    split(f,cons(bot,lu),lv) -> bot
    split(f,cons(plus(u1,u2),lu),lv) ->
        plus(Appl(f,rest(lu,cons(u1,lv))),
             Appl(f,rest(lu,cons(u2,lv))))
    split(f,nil,lv) -> frozen(f,rest(nil,lv))
    rest(lu,nil) -> lu
    rest(lu,cons(u,lv)) -> rest(cons(u,lu),lv)

  times(bot,v,sig) -> bot
  times(v,bot,sig) -> bot

  times(plus(u1,u2),v,sig) ->
        plus(times(u1,v,sig),times(u2,v,sig))
  times(v,plus(u1,u2),sig) ->
        plus(times(v,u1,sig),times(v,u2,sig))


  minus(v, var(n,bot)) -> bot
  minus(v, bot) -> v
  minus(plus(v1,v2), w) ->
        plus(minus(v1, w), minus(v2, w))

  minus(bot, appl(f,lv)) -> bot

  minus(appl(f,lu), plus(v,w)) ->
        minus(minus(appl(f,lu),v),w)

  minus(appl(f,lu), appl(f,lv)) ->
        genm7(f,lu,lv,len(lu))
    genm7(f,lu,lv,z) -> bot
    genm7(f,lu,lv,suc(i)) ->
        plus(genm7(f,lu,lv,i),
```

```
            appl(f,diff(lu,lv,suc(i))))
    diff(nil,nil,i) -> nil
    diff(cons(u,lu),cons(v,lv),s(s(i))) ->
        cons(u,diff(lu,lv,s(i)))
    diff(cons(u,lu),cons(v,lv),s(z)) ->
        cons(minus(u,v),lu)
    len(nil) -> z
    len(cons(u,lu)) -> s(len(lu))

  minus(appl(f,lu), appl(g,lv)) -> appl(f,lu)


  times(v, var(n,bot), sig) -> v
  times(var(n,bot), v, sig) -> v

  times(appl(f,lu), appl(f,lv), sig) -> dist(appl(f,nil), prod(lu,lv,nil,sig)))
   prod(cons(u,lu),cons(v,lv),lacc,sig) -> prod(lu,lv,cons(times(u,v,sig),lacc),sig)

   dist(appl(f,l), prod(nil,nil,nil,sig)) -> appl(f,l)
   dist(appl(f,l), prod(nil,nil,cons(u,lu),sig))
                  -> dist(appl(f,cons(u,l)), prod(nil,nil,lu,sig))

  times(appl(f,lu), appl(g,lv), sig) -> bot

  times(var(n,p), appl(f,lv), sig) ->
         times(gensum(sig,p),minus(appl(f,lv), p), sig)
     gensum(nilsig,p) -> bot
     gensum(conssig(f,n,tail),p) ->
         plus(appl(f,genvar(n,p)), gensum(tail,p))
     genvar(z,p) -> nil
     genvar(s(n),p) -> cons(var(s(n),p),genvar(n,p))

  times(appl(f,lu), minus(var(m,p), t), sig) -> minus(times(appl(f,lu), var(m,p), sig), t)

  times(var(n,p), minus(var(m,q), t), sig) -> minus(times(var(n,p), var(m,q), sig), t)

   times(minus(var(n,p),v), t, sig) -> minus(times(var(n,p),v,sig), t)
   minus(minus(var(n,p),v), t) -> minus(var(n,p),plus(v,t))

   minus(var(n,p),v) -> bot
)
```