

# Preservation of Pattern-free Properties for Constructor TRS

Horatiu Cirstea

Pierre Lermusiaux

Pierre-Etienne Moreau

Université de Lorraine, CNRS, Inria, Loria  
F-54000 Nancy, France

{name}.{surname}@loria.fr

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. We propose here an approach where the function symbols corresponding to the transformations performed in a pass are annotated with the patterns they are supposed to eliminate and show how we can check that the transformation is consistent with the annotations and thus, that it eliminates the respective patterns.

## 1 Introduction

Rewriting is a well established formalism widely used in both computer science and mathematics. Rewriting has turned out to be particularly well adapted to describe program semantics [23] and program transformations [20, 6]. 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 can be used to guarantee 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 proposed for NanoPass [16] 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 the corresponding intermediate language.

Let's consider expressions which are build out of (wrapped) integers, (wrapped) strings and lists:

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

If, for some reason, we want to define a pass 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:

$$\begin{array}{lll} Expr & = & lit(Literal) \quad Literal = int(Int) \quad List = nil() \\ & | & lst(List) \quad \quad \quad | \quad str(String) \quad \quad \quad | \quad cons(Literal, List) \end{array}$$

Functional approaches to transformation [22] relying on the use of fine grained typing systems which combine overloading, subtyping and polymorphism through the use of variants [11] 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 underlying rewriting system is consistent with the annotations. The method is minimally intrusive: for the above example, we should just annotate the flattening function symbol with the (anti-)pattern  $cons(lst(l_1), l_2)$  as detailed in Section 4.3.

First, in the next section, we introduce the basic notions and notations used in the article. 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 We finally present some related work and conclude.

## 2 Preliminary notions

We define here the basic notions and notations used in this paper; more details can be found in [4, 25].

A *many-sorted signature*  $\Sigma = (\mathcal{S}, \mathcal{F})$ , consists of a set of sorts  $\mathcal{S}$  and a set of symbols  $\mathcal{F}$ . We distinguish constructor symbols from function symbols by partitioning the alphabet  $\mathcal{F}$  into  $\mathcal{D}$ , the set of *defined symbols*, and  $\mathcal{C}$  the set of *constructors*. A symbol  $f$  with *domain*  $Dom(f) = s_1 \times \dots \times s_n \in \mathcal{S}^*$  and *co-domain*  $CoDom(f) = s \in \mathcal{S}$  is written  $f:s_1 \times \dots \times s_n \mapsto s$ ; 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  $f(t_1, \dots, t_n)$  is in  $\mathcal{T}_s(\mathcal{F}, \mathcal{X})$  whenever  $f:s_1 \times \dots \times s_n \mapsto s$  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  $\mathcal{T}(\mathcal{F}, \mathcal{X}) = \bigcup_{s \in \mathcal{S}} \mathcal{T}_s(\mathcal{F}, \mathcal{X})$ .  $Var(t)$  denotes the set of variables occurring in  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ . If  $Var(t)$  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  $\varepsilon$ .  $t|_\omega$ , resp.  $t(\omega)$ , denotes the subterm of  $t$ , resp. the symbol of  $t$ , at position  $\omega$ .  $t[s]_\omega$  denotes the term  $t$  with the subterm at position  $\omega$  replaced by  $s$ .  $Pos(t)$  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  $\sigma$  extends as expected to an endomorphism  $\sigma'$  of  $\mathcal{T}(\mathcal{F}, \mathcal{X})$ . To simplify the notations, we do not make the distinction between  $\sigma$  and  $\sigma'$ . 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  $\sigma$ ,  $t:s$  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  $p \ll v$ ) if it exists a substitution  $\sigma$  such that  $v = \sigma(p)$ . Since  $p$  is linear, we can give an inductive definition to the pattern matching relation:

$$\begin{aligned} x &\ll v && x \in \mathcal{X} \\ c(p_1, \dots, p_n) &\ll c(v_1, \dots, v_n) && \text{iff } \bigwedge_{i=1}^n p_i \ll v_i, \text{ for } c \in \mathcal{C} \end{aligned}$$

Starting from the observation that a pattern can be interpreted as the set of its instances, the notion of *ground semantics* was introduced in [8] as the set of all ground constructor instances of a pattern  $p \in \mathcal{T}_s(\mathcal{C}, \mathcal{X})$ :  $\llbracket p \rrbracket = \{\sigma(p) \mid \sigma(p) \in \mathcal{T}_s(\mathcal{C})\}$ . It was shown [8] that, given a pattern  $p$  and a value  $v$ ,  $v \in \llbracket p \rrbracket$  iff  $p \ll v$ . We denote by  $\perp$  the pattern whose semantics is empty, *i.e.* matching no term.

A *constructor rewrite rule* (over  $\Sigma$ ) is a pair of terms  $\varphi(l_1, \dots, l_n) \rightarrow r \in \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \times \mathcal{T}_s(\mathcal{F}, \mathcal{X})$  with  $s \in \mathcal{S}$ ,  $\varphi \in \mathcal{D}$ ,  $l_1, \dots, l_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$  and such that  $\varphi(l_1, \dots, l_n)$  is linear and  $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ . A *constructor based term rewriting system* (CBTRS) is a set of constructor rewrite rules  $\mathcal{R}$  inducing a *rewriting relation* over  $\mathcal{T}(\mathcal{F})$ , denoted by  $\rightarrow_{\mathcal{R}}$  and such that  $t \rightarrow_{\mathcal{R}} t'$  iff there exist  $l \rightarrow r \in \mathcal{R}$ ,  $\omega \in \mathcal{P}os(t)$ , and a substitution  $\sigma$  such that  $t_{l\omega} = \sigma(l)$  and  $t' = t[\sigma(r)]_{\omega}$ . The reflexive and transitive closure of  $\rightarrow_{\mathcal{R}}$  is denoted by  $\rightarrow^*_{\mathcal{R}}$ .

### 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 be always 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-free properties and subsequently verify the consistency of the symbol annotations with a given 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, a ground term of the form  $f^{-p}(t_1, \dots, t_n)$  should ultimately be reduced to a value containing no subterms matched by  $p$ . Given the example given in the introduction, we can consider two function symbols,  $flattenE^{-p} : Expr \mapsto Expr$  and  $flattenL^{-p} : List \mapsto List$ , with  $p = cons(lst(l_1), l_2)$ , to indicate that the normal forms of any term headed by one of these symbols contains no nested lists. The annotation of the function symbol for the concatenation,  $concat^{-\perp} : List \times List \mapsto List$ , indicates that no particular shape is expected for the reducts of the corresponding terms. For readability, we generally omit the  $\perp$  pattern annotation.

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

- a value  $v \in \mathcal{T}(\mathcal{C})$  is  $p$ -free iff  $\forall \omega \in \mathcal{Pos}(v), p \not\prec v|_{\omega}$ ;
- a pattern  $t \in \mathcal{T}(\mathcal{C}, \mathcal{X})$  is  $p$ -free iff  $\forall \sigma$  such that  $\sigma(t) \in \mathcal{T}(\mathcal{C})$ ,  $\sigma(t)$  is  $p$ -free.
- a term  $t[f_s^{-q}(t_1, \dots, t_n)]_{\omega} \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  is  $p$ -free iff  $t[v]_{\omega}$  is  $p$ -free  $\forall v \in \mathcal{T}_s(\mathcal{C})$   $q$ -free;

A value is  $p$ -free if and only if  $p$  matches no subterm of the value. For constructor patterns, verifying a pattern-free property comes to verifying the property for all the ground instances of the term. Finally, a general term is  $p$ -free if and only if 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 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 4.2 an approach for solving this problem.

### 3.2 Generalized ground semantics

The notion of ground semantics presented in Section 2 and the approach proposed in [8] 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  $\llbracket p \rrbracket \cap \llbracket q \rrbracket = \emptyset$  we have that  $\forall \sigma, \sigma(q) \notin \llbracket p \rrbracket$  and therefore, we can establish that  $\forall \sigma, p \not\prec \sigma(q)$ . 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  $\Sigma$  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  $\llbracket c_1(c_4(c_6), y_{s_1}) \rrbracket \cap \llbracket c_1(x_{s_2}, c_2(c_6)) \rrbracket = \llbracket c_1(c_4(c_6), c_2(c_6)) \rrbracket$  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  $\llbracket c_3(c_2(z_{s_3})) \rrbracket \cap \llbracket c_4(z_{s_3}) \rrbracket = \emptyset$  and  $\llbracket c_2(z_{s_3}) \rrbracket \cap \llbracket c_4(z_{s_3}) \rrbracket = \emptyset$  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 introduced in [8] extended to take into account all terms in  $\mathcal{T}(\mathcal{F}, \mathcal{X})$ :

**Definition 3.2** (Generalized ground semantics). *Given a sort  $s \in \mathcal{S}$ , a pattern  $p \in \mathcal{T}_{\perp}(\mathcal{C}, \mathcal{X})$ , a symbol  $f_s^{-P} : s_1 \times \dots \times s_n \mapsto s \in \mathcal{D}_s$ , a pattern  $u \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ , and a term  $t[f_s^{-P}(t_1, \dots, t_n)]_{\omega} \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  with  $t_i \in \mathcal{T}_{s_i}(\mathcal{F}, \mathcal{X}), i \in [1, n]$ ,*

- $\llbracket u \rrbracket = \{\sigma(u) \mid \sigma(u) \in \mathcal{T}(\mathcal{C})\}$ ;
- $\llbracket t[f_s^{-P}(t_1, \dots, t_n)]_{\omega} \rrbracket = \{u \in \llbracket t[v]_{\omega} \rrbracket \mid \forall v \in \mathcal{T}_s(\mathcal{C}) \text{ } p\text{-free}\}$ .

Note that the ground semantics of a variable  $x_s$  is the set of all possible ground patterns of the corresponding sort:  $\llbracket x_s \rrbracket = \mathcal{T}_s(\mathcal{C})$ , and since patterns are linear we can use a recursive definition for the non-variable patterns:  $\llbracket c(p_1, \dots, p_n) \rrbracket = \{c(v_1, \dots, v_n) \mid (v_1, \dots, v_n) \in \llbracket p_1 \rrbracket \times \dots \times \llbracket p_n \rrbracket\}$  for all  $c \in \mathcal{C}$ . Moreover, it is easy to check that  $\llbracket f_s^{-P}(t_1, \dots, t_n) \rrbracket = \{v \in \mathcal{T}_s(\mathcal{C}) \mid v \text{ } p\text{-free}\}$ .

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 TRS preserving the pattern-free properties. 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:

$$\llbracket x_s^{-P} \rrbracket = \{v \mid v \in \mathcal{T}_s(\mathcal{C}) \wedge v \text{ } p\text{-free}\}$$

Thus,  $\llbracket f_s^{-P}(t_1, \dots, t_n) \rrbracket = \llbracket x_s^{-P} \rrbracket, \forall f_s^{-P} \in \mathcal{D}_s$ . Note that  $x_s^{-\perp}$  has the same semantics as  $x_s$ . We denote by  $\mathcal{X}^a$  the set of annotated variables.

Moreover, given a 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 of the corresponding sort and annotated by the same pattern:  $\forall t \in \mathcal{T}(\mathcal{F}, \mathcal{X}), \llbracket t \rrbracket = \llbracket \tilde{t} \rrbracket$ . We can thus restrict in what follows to patterns using annotated variables and we consider *extended patterns* built out of this kind of patterns:

$$p := \mathcal{X}^a \mid c(q_1, \dots, q_n) \mid p_1 + p_2 \mid p_1 \setminus p_2 \mid p_1 \times p_2 \mid \perp$$

with  $p, p_1, p_2 : s$  for some  $s \in \mathcal{S}, c : s_1 \times \dots \times s_n \mapsto s \in \mathcal{C}$  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:

$$\begin{array}{ll} p_1 + p_2 \llcorner v & \text{iff } p_1 \llcorner v \vee p_2 \llcorner v & p_1 \times p_2 \llcorner v & \text{iff } p_1 \llcorner v \wedge p_2 \llcorner v \\ p_1 \setminus p_2 \llcorner v & \text{iff } p_1 \llcorner v \wedge p_2 \not\llcorner v & \perp \llcorner v & \text{iff } \text{false} \end{array}$$

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  $p_2$ .  $\perp$  matches no term.  $\times$  has a higher priority than  $\setminus$  which has a higher priority than  $+$ .

The notion of ground semantics can be also adapted to handle such patterns:

$$\begin{array}{ll} \llbracket p_1 + p_2 \rrbracket = \llbracket p_1 \rrbracket \cup \llbracket p_2 \rrbracket & \llbracket p_1 \setminus p_2 \rrbracket = \llbracket p_1 \rrbracket \setminus \llbracket p_2 \rrbracket \\ \llbracket p_1 \times p_2 \rrbracket = \llbracket p_1 \rrbracket \cap \llbracket p_2 \rrbracket & \llbracket \perp \rrbracket = \emptyset \end{array}$$

In this context, if an extended pattern contains no  $\perp$  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  $x^{-\perp}$ . 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.

**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  $\forall v \in \llbracket t \rrbracket, v$  is  $p$ -free.*

Pattern-freeness can thus be checked by exploring the semantics of the term. We will see later on how this can be done and for now we focus on semantics preserving CBTRSs and on their properties.

### 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** (Semantics preservation). *A rewrite rule  $l \rightarrow r$  is semantics preserving iff  $\llbracket r \rrbracket \subseteq \llbracket l \rrbracket$ . A CBTRS is semantics preserving iff all its rewrites rules are.*

Semantics preservation carries over to the induced rewriting relation:

**Proposition 3.2.** *Given a semantics preserving CBTRS  $\mathcal{R}$  we have*

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

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

**Corollary 3.1.** *Given a semantics preserving CBTRS  $\mathcal{R}$  we have*

$$\forall t, v \in \mathcal{T}(\mathcal{F}), p \in \mathcal{T}(\mathcal{C}, \mathcal{X}), \text{ if } t \text{ } p\text{-free and } t \twoheadrightarrow_{\mathcal{R}} v, \text{ then } v \text{ } p\text{-free.}$$

Note that the rules of a CBTRS are of the form  $f^{-p}(l_1, \dots, l_n) \rightarrow r$  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 can be statically checked.

**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  $p_2 = c_4(z)$ , and the CBTRS:*

$$\begin{array}{ll} f(c_1(x, y)) \rightarrow c_1(g(x), f(y)) & g(c_4(z)) \rightarrow c_3(c_2(z)) \\ f(c_2(z)) \rightarrow c_2(z) & g(c_3(y)) \rightarrow c_3(f(y)) \end{array}$$

*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  $g(c_4(z)) \rightarrow c_3(c_2(z))$  is semantics preserving. We introduce in the next section a method to statically check the corresponding pattern-freeness for the other right-hand sides and conclude that the CBTRS is semantics preserving.*

## 4 Deep semantics for pattern-free properties

The ground semantics was used in [8] as a means to represent a potentially infinite number of instances of a term in a finite manner and to subsequently check, for example, 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 need thus 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 comparing 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  $\llbracket t \rrbracket$  is as:*

$$\llbracket t \rrbracket = \{u|_{\omega} \mid u \in \llbracket t \rrbracket, \omega \in \mathcal{P}os(u)\}$$

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.*  $\forall t \in \mathcal{T}(\mathcal{F}, \mathcal{X}), \llbracket t \rrbracket = \llbracket \tilde{t} \rrbracket$ ; 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.** *Let  $p \in \mathcal{T}(\mathcal{C}, \mathcal{X}), t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ ,  $t$  is  $p$ -free iff  $\llbracket \tilde{t} \rrbracket \cap \llbracket p \rrbracket = \emptyset$ .*

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  $t_1, \dots, t_n$ , such that  $\text{Dom}(c) = s_1 \times \dots \times s_n$  and  $t_1 : s_1, \dots, t_n : s_n$ , we have:*

- *If  $\forall i \in [1, n], \llbracket t_i \rrbracket \neq \emptyset$ , then  $\llbracket c(t_1, \dots, t_n) \rrbracket = \llbracket c(t_1, \dots, t_n) \rrbracket \cup \left( \bigcup_{i=1}^n \llbracket t_i \rrbracket \right)$ ;*
- *If  $\exists i \in [1, n], \llbracket t_i \rrbracket = \emptyset$ , then  $\llbracket c(t_1, \dots, t_n) \rrbracket = \emptyset$ .*

If we apply the above equation for the non-empty case recursively we eventually have to compute the deep semantics of annotated variables and for this we use the algorithm `getReachable` given in Appendix A.

**Proposition 4.3.** *Given  $s \in \mathcal{S}, p \in \mathcal{T}_\perp(\mathcal{C}, \mathcal{X})$  and  $r : s$  a sum of constructor patterns, we have*

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

with  $R = \text{getReachable}(s, p, \emptyset, r)$ . Moreover, we have  $\llbracket x_s^{-p} \setminus r \rrbracket = \emptyset$  iff  $R = \emptyset$ .

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

As we have already mentioned, an approach was proposed in [8] to compute the intersection of ground semantics for constructor patterns. In our case we just need to check pattern-freeness and thus, that the intersection of the semantics of a symbolic pattern  $t$  with the semantics of the given constructor pattern  $p$  is empty: to put it simply, 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 1. The rules generally correspond to their counterparts from set theory where  $\perp$  corresponds to the empty set, a variable  $x_s^{-\perp}$  corresponds to all terms of sort  $s$ , constructor patterns correspond to cartesian products and the other extended patterns to the obvious corresponding set operations.

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 `getReachable` we can determine if  $\llbracket \bar{x}_s^{-\bar{p}} \setminus \bar{v} \rrbracket = \emptyset$ . Moreover, by definition,  $\llbracket t \rrbracket = \emptyset$  if and only if  $\llbracket t \rrbracket = \emptyset$ . Therefore, the TRS is finalized by the rule P6, which eliminates (when possible) annotated variables. Note that, in order to apply P6 exhaustively,  $\mathcal{R}_p$  also needs a rule to perform some  $\setminus$ -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  $\llbracket p \rrbracket = \llbracket q \rrbracket$ .*

While we cannot provide a simple description of the normal forms obtained by reduction of a general extended patterns,  $\mathcal{R}_p$  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  $t \times p \twoheadrightarrow_{\mathcal{R}_p} \perp$  if and only if  $\llbracket t \times p \rrbracket = \emptyset$ .*

<b>Remove empty sets:</b>	
(A1)	$\perp + \bar{v} \Rightarrow \bar{v}$
(A2)	$\bar{v} + \perp \Rightarrow \bar{v}$
<b>Distribute sets:</b>	
(E1)	$\delta(\bar{v}_1, \dots, \perp, \dots, \bar{v}_n) \Rightarrow \perp$
(E2)	$\perp \times \bar{v} \Rightarrow \perp$
(E3)	$\bar{v} \times \perp \Rightarrow \perp$
(S1)	$\delta(\bar{v}_1, \dots, \bar{v}_i + \bar{w}_i, \dots, \bar{v}_n) \Rightarrow \delta(\bar{v}_1, \dots, \bar{v}_i, \dots, \bar{v}_n) + \delta(\bar{v}_1, \dots, \bar{w}_i, \dots, \bar{v}_n)$
(S2)	$(\bar{w}_1 + \bar{w}_2) \times \bar{v} \Rightarrow (\bar{w}_1 \times \bar{v}) + (\bar{w}_2 \times \bar{v})$
(S3)	$\bar{w} \times (\bar{v}_1 + \bar{v}_2) \Rightarrow (\bar{w} \times \bar{v}_1) + (\bar{w} \times \bar{v}_2)$
<b>Simplify complements:</b>	
(M1)	$\bar{v} \setminus \bar{x}_s^{-\perp} \Rightarrow \perp$
(M2)	$\bar{v} \setminus \perp \Rightarrow \bar{v}$
(M3)	$(\bar{v}_1 + \bar{v}_2) \setminus \bar{w} \Rightarrow (\bar{v}_1 \setminus \bar{w}) + (\bar{v}_2 \setminus \bar{w})$
(M5)	$\perp \setminus \bar{v} \Rightarrow \perp$
(M6)	$\alpha(\bar{v}_1, \dots, \bar{v}_n) \setminus (\bar{v} + \bar{w}) \Rightarrow (\alpha(\bar{v}_1, \dots, \bar{v}_n) \setminus \bar{v}) \setminus \bar{w}$
(M7)	$\alpha(\bar{v}_1, \dots, \bar{v}_n) \setminus \alpha(\bar{t}_1, \dots, \bar{t}_n) \Rightarrow \alpha(\bar{v}_1 \setminus \bar{t}_1, \dots, \bar{v}_n) + \dots + \alpha(\bar{v}_1, \dots, \bar{v}_n \setminus \bar{t}_n)$
(M8)	$\alpha(\bar{v}_1, \dots, \bar{v}_n) \setminus \beta(\bar{w}_1, \dots, \bar{w}_m) \Rightarrow \alpha(\bar{v}_1, \dots, \bar{v}_n) \quad \text{with } \alpha \neq \beta$
<b>Simplify conjunctions:</b>	
(T1)	$\bar{v} \times \bar{x}_s^{-\perp} \Rightarrow \bar{v}$
(T2)	$\bar{x}_s^{-\perp} \times \bar{v} \Rightarrow \bar{v}$
(T3)	$\alpha(\bar{v}_1, \dots, \bar{v}_n) \times \alpha(\bar{w}_1, \dots, \bar{w}_n) \Rightarrow \alpha(\bar{v}_1 \times \bar{w}_1, \dots, \bar{v}_n \times \bar{w}_n)$
(T4)	$\alpha(\bar{v}_1, \dots, \bar{v}_n) \times \beta(\bar{w}_1, \dots, \bar{w}_m) \Rightarrow \perp \quad \text{with } \alpha \neq \beta$
<b>Simplify p-free:</b>	
(P1)	$\bar{x}_s^{-\bar{p}} \times \alpha(\bar{v}_1, \dots, \bar{v}_n) \Rightarrow \sum_{c \in \mathcal{C}_s} c(z_{1s_1}^{-\bar{p}}, \dots, z_{ms_m}^{-\bar{p}}) \times (\alpha(\bar{v}_1, \dots, \bar{v}_n) \setminus \bar{p}) \quad \text{with } m = \text{arity}(c)$
(P2)	$\alpha(\bar{v}_1, \dots, \bar{v}_n) \times (\bar{x}_s^{-\bar{p}} \setminus \bar{t}) \Rightarrow (\alpha(\bar{v}_1, \dots, \bar{v}_n) \times \bar{x}_s^{-\bar{p}}) \setminus \bar{t} \quad \text{if } \{\bar{x}_s^{-\bar{p}} \setminus \bar{t}\} \neq \emptyset$
(P3)	$\bar{x}_s^{-\bar{q}} \times (\bar{x}_s^{-\bar{p}} \setminus \bar{t}) \Rightarrow (\bar{x}_s^{-\bar{q}} \times \bar{x}_s^{-\bar{p}}) \setminus \bar{t} \quad \text{if } \{\bar{x}_s^{-\bar{p}} \setminus \bar{t}\} \neq \emptyset$
(P4)	$(\bar{x}_s^{-\bar{p}} \setminus \bar{t}) \times \bar{v} \Rightarrow (\bar{x}_s^{-\bar{p}} \times \bar{v}) \setminus \bar{t} \quad \text{if } \{\bar{x}_s^{-\bar{p}} \setminus \bar{t}\} \neq \emptyset$
(P5)	$(\bar{x}_s^{-\bar{p}} \setminus \bar{t}) \setminus \bar{u} \Rightarrow \bar{x}_s^{-\bar{p}} \setminus (\bar{t} + \bar{u}) \quad \text{if } \{\bar{x}_s^{-\bar{p}} \setminus \bar{t}\} \neq \emptyset$
(P6)	$\bar{x}_s^{-\bar{p}} \setminus \bar{t} \Rightarrow \perp \quad \text{if } \{\bar{x}_s^{-\bar{p}} \setminus \bar{t}\} = \emptyset$

Figure 1:  $\mathcal{R}_p$ : reduce pattern of the form  $t \times p$ ;  $\bar{v}, \bar{v}_1, \dots, \bar{v}_n, \bar{w}, \bar{w}_1, \dots, \bar{w}_n$  range over quasi-additive patterns,  $\bar{u}, \bar{t}$  range over pure regular additive patterns,  $\bar{t}_1, \dots, \bar{t}_n$  range over pure symbolic patterns,  $\bar{p}, \bar{q}$  range over constructor patterns,  $\bar{x}$  ranges over pattern variables.  $\alpha, \beta$  expand to all the symbols in  $\mathcal{C}$ ,  $\delta$  expands to all symbols in  $\mathcal{C}^{n>0}$ .

### 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  $\mathcal{T}(\mathcal{F}, \mathcal{X})$  and can thus be used to check that a CBTRS is semantics preserving.

**Example 4.1.** We apply the approach to check that the CBTRS in Example 3.2 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  $r_1 = c_1(g^{-p_2}(x_{s_2}), f^{-p_1}(y_{s_1}))$ , it is  $p_1$ -free, we first use `getReachable` to compute the deep semantics of  $\tilde{r}_1 = c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})$  as the union of  $\llbracket c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1}) \rrbracket$ ,  $\llbracket y_{s_1}^{-p_1} \rrbracket$ ,  $\llbracket z_{s_3}^{-p_1} \rrbracket$ ,  $\llbracket x_{s_2}^{-p_1} \setminus p_2 \rrbracket$ ,  $\llbracket x_{s_2}^{-p_2} \rrbracket$ ,  $\llbracket y_{s_1}^{-p_2} \rrbracket$  and  $\llbracket z_{s_3}^{-p_2} \rrbracket$ . For all the terms in the union we compute their conjunction with  $p_1$  using  $\mathcal{R}_p$  which reduces them all to  $\perp$ . 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  $\mathcal{D} = \{flattenE^{-p} : Expr \mapsto Expr, flattenL^{-p} : List \mapsto List, concat^{-\perp} : List \times List \mapsto List\}$ . with  $p = conc(lst(l_1), l_2)$ , to indicate that the corresponding functions defined by the following CBTRS aim at eliminating this pattern:

$$\left\{ \begin{array}{ll} flattenE(str(s)) & \rightarrow str(s) \\ flattenE(lst(l)) & \rightarrow lst(flattenL(l)) \\ flattenL(nil()) & \rightarrow nil() \\ flattenL(cons(str(s), l)) & \rightarrow cons(str(s), flattenL(l)) \\ flattenL(cons(lst(l_1), l_2)) & \rightarrow flattenL(concat(l_1, l_2)) \\ concat(cons(e, l_1), l_2) & \rightarrow cons(e, concat(l_1, l_2)) \\ concat(nil(), l) & \rightarrow l \end{array} \right.$$

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 form of such terms are  $p$ -free values.

The implementation<sup>1</sup> takes an 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.

## 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, other 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 [12]. The application of this approach is nevertheless usually conditioned by the termination of both the TRS and the set of equational approximations used [24, 13]. Thus, while providing a more precise characterization of the approximations of the normal forms, these techniques are constrained by these termination conditions. Therefore we believe the formalism presented in this paper provides a viable and original alternative to such techniques, particularly in the context of verification of pass transformations [16].

<sup>1</sup>code available at [https://github.com/plermusiaux/pfree\\_check](https://github.com/plermusiaux/pfree_check); online version available at [http://htmlpreview.github.io/?https://github.com/plermusiaux/pfree\\_check/blob/webnix/out/index.html](http://htmlpreview.github.io/?https://github.com/plermusiaux/pfree_check/blob/webnix/out/index.html)

**Recursion schemes** Some formalism 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 [17]. Higher order recursion scheme have also been extended to include pattern matching [21] 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 terms rewriting systems, another popular approach for specifying transformations consists in the use of tree transducers [18]. 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 [18] 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 [9] can be seen as an alternative approach for verifying the absence, or presence, of specific patterns. Notions such as constructor subtypes [5] 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 [16].

## 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 an intermediate language for each of the so-called passes in a compilation process, the user being just requested to indicate the patterns that should be eliminated by the respective transformation.

Different termination analysis techniques [3, 15, 14] and corresponding tools like AProVE [10] and TTT2 [19] can be used for checking the termination of the rewriting systems before applying our method for checking pattern-free properties. On the other hand, our 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.

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, if in the CBTRS in Section 4.3 we replace the right-hand side of the third rule for *flattenL* by  $concat(flattenL(l_1), flattenL(l_2))$  then the rule would be reported as non pattern-preserving. Although we conjecture that such false negatives indicate some issues in the conception of the CBTRS, we work on an alternative approach allowing for a finer-grain analysis. We also intend to extend and use the approach in the context of automatic rewriting rule generation techniques, such as the one introduced in [7], in order to implement transformation approaches of passes such as in [16].

## References

- [1] Rajeev Alur & Pavol Cerný (2011): *Streaming transducers for algorithmic verification of single-pass list-processing programs*. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, ACM, pp. 599–610, doi:10.1145/1926385.1926454.
- [2] Rajeev Alur, Emmanuel Filiot & Ashutosh Trivedi (2012): *Regular Transformations of Infinite Strings*. In: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012*, IEEE Computer Society, pp. 65–74, doi:10.1109/LICS.2012.18.
- [3] Thomas Arts & Jürgen Giesl (2000): *Termination of term rewriting using dependency pairs*. *Theoretical Computer Science* 236(1-2), pp. 133–178, doi:10.1016/S0304-3975(99)00207-8.
- [4] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press.
- [5] Gilles Barthe & Maria João Frade (1999): *Constructor Subtyping*. In: *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Lecture Notes in Computer Science 1576*, Springer, pp. 109–127, doi:10.1007/3-540-49099-X\_8.
- [6] Françoise Bellegarde (1991): *Program Transformation and Rewriting*. In: *Rewriting Techniques and Applications, 4th International Conference, RTA-91, Lecture Notes in Computer Science 488*, Springer, pp. 226–239, doi:10.1007/3-540-53904-2\_99.
- [7] Horatiu Cirstea, Sergueï Lenglet & Pierre-Etienne Moreau (2015): *A faithful encoding of programmable strategies into term rewriting systems*. In: *26th International Conference on Rewriting Techniques and Applications, RTA 2015, LIPIcs 36*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 74–88, doi:10.4230/LIPIcs.RTA.2015.74.
- [8] Horatiu Cirstea & Pierre-Etienne Moreau (2019): *Generic Encodings of Constructor Rewriting Systems*. In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019*, ACM, pp. 8:1–8:12, doi:10.1145/3354166.3354173.
- [9] Timothy S. Freeman & Frank Pfenning (1991): *Refinement Types for ML*. In: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, ACM, pp. 268–277, doi:10.1145/113445.113468.
- [10] Carsten Fuhs, Jürgen Giesl, Michael Parting, Peter Schneider-Kamp & Stephan Swiderski (2011): *Proving Termination by Dependency Pairs and Inductive Theorem Proving*. *Journal of Automatic Reasoning* 47(2), pp. 133–160, doi:10.1007/s10817-010-9215-9.
- [11] Jacques Garrigue (1998): *Programming with polymorphic variants*. In: *ACM Workshop on ML*.
- [12] Thomas Genet (2014): *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 8663*, Springer, pp. 147–161, doi:10.1007/978-3-319-12904-4\_8.
- [13] Thomas Genet (2016): *Termination criteria for tree automata completion*. *Journal of Logical and Algebraic Methods in Programming* 85(1), pp. 3–33, doi:10.1016/j.jlamp.2015.05.003.
- [14] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp & Stephan Falke (2006): *Mechanizing and Improving Dependency Pairs*. *Journal of Automatic Reasoning* 37(3), pp. 155–203, doi:10.1007/s10817-006-9057-7.
- [15] Nao Hirokawa & Aart Middeldorp (2005): *Automating the dependency pair method*. *Information and Computation* 199(1-2), pp. 172–199, doi:10.1016/j.ic.2004.10.004.
- [16] Andrew W. Keep & R. Kent Dybvig (2013): *A nanopass framework for commercial compiler development*. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13*, ACM, pp. 343–350, doi:10.1145/2500365.2500618.
- [17] Naoki Kobayashi (2009): *Types and higher-order recursion schemes for verification of higher-order programs*. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, ACM, pp. 416–428, doi:10.1145/1480881.1480933.

- [18] Naoki Kobayashi, Naoshi Tabuchi & Hiroshi Unno (2010): *Higher-order multi-parameter tree transducers and recursion schemes for program verification*. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, ACM, pp. 495–508, doi:10.1145/1706299.1706355.
- [19] Martin Korp, Christian Sternagel, Harald Zankl & Aart Middeldorp (2009): *Tyrolean Termination Tool 2*. In Ralf Treinen, editor: *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Lecture Notes in Computer Science 5595*, Springer, pp. 295–304, doi:10.1007/978-3-642-02348-4\_21.
- [20] David Lacey & Oege de Moor (2001): *Imperative Program Transformation by Rewriting*. In Reinhard Wilhelm, editor: *Compiler Construction, 10th International Conference, CC 2001, Lecture Notes in Computer Science 2027*, Springer, pp. 52–68, doi:10.1007/3-540-45306-7\_5.
- [21] C.-H. Luke Ong & Steven J. Ramsay (2011): *Verifying higher-order functional programs with pattern-matching algebraic data types*. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, ACM, pp. 587–598, doi:10.1145/1926385.1926453.
- [22] François Pottier (2017): *Visitors unchained*. *Proceedings of the ACM on Programming Languages* 1(ICFP), pp. 28:1–28:28, doi:10.1145/3110272.
- [23] Grigore Rosu & Traian-Florin Serbanuta (2010): *An overview of the K semantic framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.
- [24] Toshinori Takai (2004): *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 3091*, Springer, pp. 119–133, doi:10.1007/978-3-540-25979-4\_9.
- [25] Terese (2003): *Term Rewriting Systems*. Cambridge University Press. M. Bezem, J. W. Klop and R. de Vrijer, eds.

## A Algorithm getReachable

The deep semantics of annotated variables can be computed using the algorithm `getReachable` introduced in Figure 2.

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

$$\llbracket x_s^{-p} \rrbracket = \bigcup_{c \in \mathcal{C}_s} \llbracket c(x_{s_1}^{-p}, \dots, x_{s_i}^{-p}) \setminus p \rrbracket \quad (1)$$

By distributing the complement pattern  $p$  on the subterms, the algorithm builds a set  $\mathcal{Q}_c(p)$  of tuples  $q = (q_1, \dots, q_n)$  of patterns, with each  $q_i$  being either  $\perp$  or a subterm of  $p$ , such that

$$\llbracket c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \setminus p \rrbracket = \bigcup_{q \in \mathcal{Q}_c(p)} \llbracket c(x_{s_1}^{-p} \setminus q_1, \dots, x_{s_n}^{-p} \setminus q_n) \rrbracket \quad (2)$$

We have thus

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

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

Note that  $x_s^{-p}$  is the same as  $x_s^{-p} \setminus \perp$  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

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

**Example A.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  $\llbracket \tilde{r}_1 \rrbracket = \llbracket c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1}) \rrbracket = \llbracket c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1}) \rrbracket \cup \llbracket x_{s_2}^{-p_2} \rrbracket \cup \llbracket y_{s_1}^{-p_1} \rrbracket$  and we should expand  $\llbracket x_{s_2}^{-p_2} \rrbracket$  and  $\llbracket y_{s_1}^{-p_1} \rrbracket$ .

To expand  $\llbracket y_{s_1}^{-p_1} \rrbracket$  the sets  $\mathcal{Q}_c(p_1)$  are computed for each  $c \in \mathcal{C}_{s_1} = \{c_1, c_2\}$ . First, following equation (1),  $\llbracket y_{s_1}^{-p_1} \rrbracket = \llbracket c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1}) \setminus c_1(c_4(z_{s_3}^{-\perp}), y_{s_1}^{-\perp}) \rrbracket \cup \llbracket c_2(\bar{z}_{s_3}^{-p_1}) \setminus c_1(c_4(z_{s_3}^{-\perp}), y_{s_1}^{-\perp}) \rrbracket$  and we can easily see that the complement relation in term of ground semantics corresponds to set differences of cartesian products:  $\llbracket c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1}) \setminus c_1(c_4(z_{s_3}^{-\perp}), y_{s_1}^{-\perp}) \rrbracket = \llbracket c_1(x_{s_2}^{-p_1} \setminus c_4(z_{s_3}^{-\perp}), y_{s_1}^{-p_1}) \rrbracket \cup \llbracket c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1} \setminus y_{s_1}^{-\perp}) \rrbracket$ . We get thus,  $\llbracket y_{s_1}^{-p_1} \rrbracket = \llbracket c_1(x_{s_2}^{-p_1} \setminus c_4(z_{s_3}^{-\perp}), y_{s_1}^{-p_1}) \rrbracket \cup \llbracket c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1} \setminus y_{s_1}^{-\perp}) \rrbracket \cup \llbracket c_2(\bar{z}_{s_3}^{-p_1}) \rrbracket$ . Hence, following equation (2),  $\mathcal{Q}_{c_1}(p_1) = \{(c_4(z_{s_3}^{-\perp}), \perp), (\perp, y_{s_1})\} = \{(p_2, \perp), (\perp, y_{s_1})\}$  and  $\mathcal{Q}_{c_2}(p_1) = \{(\perp)\}$ . Moreover,  $\llbracket c_1(x_{s_2}^{-p_1} \setminus p_2, y_{s_1}^{-p_1}) \rrbracket$  and  $\llbracket c_2(\bar{z}_{s_3}^{-p_1}) \rrbracket$  are not empty ( $c_1(c_3(c_2(c_6)), c_2(c_6))$  and  $c_2(c_6)$  belong respectively to each of them) while  $\llbracket c_1(x_{s_2}^{-p_1}, y_{s_1}^{-p_1} \setminus y_{s_1}^{-\perp}) \rrbracket$  is clearly empty. Thus, we have

$$\llbracket y_{s_1}^{-p_1} \rrbracket = \llbracket y_{s_1}^{-p_1} \rrbracket \cup \llbracket x_{s_2}^{-p_1} \setminus p_2 \rrbracket \cup \llbracket y_{s_1}^{-p_1} \rrbracket \cup \llbracket \bar{z}_{s_3}^{-p_1} \rrbracket.$$

```

Function getReachable( $s, p, S, r$ )
   $s$ : current sort,
  Data:
     $p$ : pattern of the pattern-free property,
     $S$ : set of couples  $(s', p')$  reached,
     $r$ : resulting pattern
  Result: set of couples  $(s', p')$  reachable from  $x_s^{-p} \setminus r$ 
  if  $p : s$  then  $r \leftarrow r + p$ 
  if  $\llbracket x_s \setminus r \rrbracket = \emptyset$  then return  $\emptyset$ 
  if  $\exists (s, r') \in S, \llbracket r' \rrbracket = \llbracket r \rrbracket$  then return  $S$ 
   $R \leftarrow S \cup \{(s, r)\}$ 
   $reachable \leftarrow False$ 
  for  $c \in \mathcal{C}_s$  do
     $Q_c \leftarrow \{(\perp, \dots, \perp)\}$  with  $m = \text{arity}(c)$ 
    for  $i = 1$  to  $n$  with  $r = \sum_{i=1}^n r_i$  do
      if  $r_i(\omega) = c$  then
         $tQ_c \leftarrow \emptyset$ 
        for  $(q_1, \dots, q_m) \in Q_c, k \in [1, m]$  do  $tQ_c.add((q_1, \dots, q_k + r_{i|k}, \dots, q_m))$ 
         $Q_c \leftarrow tQ_c$ 
      for  $(q_1, \dots, q_m) \in Q_c$  do
         $subRset \leftarrow \emptyset$ 
        for  $i = 1$  to  $m$  do
           $subR \leftarrow \text{getReachable}(\text{Dom}(c)[i], p, R, q_i)$ 
          if  $subR \neq \emptyset$  then  $subRset.add(subR)$ 
        if  $|subRset| = m$  then
           $reachable \leftarrow True$ 
          for  $subR \in subRset$  do  $R \leftarrow R \cup subR$ 
    if  $reachable$  then
      return  $R$ 
    else
      return  $\emptyset$ 

```

Figure 2: Compute deep semantics of quasi-additive terms as a union of ground semantics.

The *getReachable* algorithm continues the expansions until a fixpoint is reached. More precisely, we get  $\llbracket y_{s_1}^{-p_1} \rrbracket = \llbracket y_{s_1}^{-p_1} \rrbracket \cup \llbracket z_{s_3}^{-p_1} \rrbracket \cup \llbracket x_{s_2}^{-p_1} \setminus p_2 \rrbracket$  and  $\llbracket x_{s_2}^{-p_2} \rrbracket = \llbracket x_{s_2}^{-p_2} \rrbracket \cup \llbracket y_{s_1}^{-p_2} \rrbracket \cup \llbracket z_{s_3}^{-p_2} \rrbracket$ , 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  $\llbracket c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1}) \rrbracket$ ,  $\llbracket y_{s_1}^{-p_1} \rrbracket$ ,  $\llbracket z_{s_3}^{-p_1} \rrbracket$ ,  $\llbracket x_{s_2}^{-p_1} \setminus p_2 \rrbracket$ ,  $\llbracket x_{s_2}^{-p_2} \rrbracket$ ,  $\llbracket y_{s_1}^{-p_2} \rrbracket$  and  $\llbracket z_{s_3}^{-p_2} \rrbracket$ .

## 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  $\forall v \in \llbracket t \rrbracket$ ,  $v$  is  $p$ -free.*

*Proof.* If  $t \in \mathcal{T}(\mathcal{C})$ , then  $\llbracket t \rrbracket = \{t\}$ , 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  $\forall v \in \llbracket t \rrbracket$ ,  $v$  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. We suppose that  $\forall u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  with  $k \leq n$  defined symbols,  $u$  is  $p$ -free iff  $\forall v \in \llbracket u \rrbracket$ ,  $v$  is  $p$ -free. We now consider  $t = u[f^{-q}(t_1, \dots, t_m)]_\omega \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  with  $n + 1$  defined symbols and  $f \in \mathcal{D}_s$ . By definition,  $t$  is  $p$ -free iff  $\forall v \in \mathcal{T}_s(\mathcal{C})$   $q$ -free,  $u[v]_\omega$  is  $p$ -free. Moreover,  $\forall v \in \mathcal{T}_s(\mathcal{C})$   $q$ -free, the number of defined symbols in  $u[v]_\omega$  is less or equal to  $n$ , thus the inductive property guarantees that  $u[v]_\omega$  is  $p$ -free iff  $\forall w \in \llbracket u[v]_\omega \rrbracket$  is  $p$ -free. Hence,  $t$  is  $p$ -free iff  $\forall w \in \{w \in \llbracket u[v]_\omega \rrbracket \mid \forall v \in \mathcal{T}_s(\mathcal{C}) \text{ } q\text{-free}, w \text{ is } p\text{-free}\}$ ,  $w$  is  $p$ -free, i.e. iff  $\forall w \in \llbracket t \rrbracket$ ,  $w$  is  $p$ -free. Hence, by induction  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ , is  $p$ -free iff  $\forall v \in \llbracket t \rrbracket$ ,  $v$  is  $p$ -free.  $\square$

**Proposition 3.2.** *Given a semantics preserving CBTRS  $\mathcal{R}$  we have*

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

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

**Proposition 4.1.** *Let  $p \in \mathcal{T}(\mathcal{C}, \mathcal{X}), t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ ,  $t$  is  $p$ -free iff  $\{\tilde{t}\} \cap \llbracket p \rrbracket = \emptyset$ .*

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

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

- If  $\forall i \in [1, n], \llbracket t_i \rrbracket \neq \emptyset$ , then  $\llbracket c(t_1, \dots, t_n) \rrbracket = \llbracket c(t_1, \dots, t_n) \rrbracket \cup \left( \bigcup_{i=1}^n \llbracket t_i \rrbracket \right)$ ;
- If  $\exists i \in [1, n], \llbracket t_i \rrbracket = \emptyset$ , then  $\llbracket c(t_1, \dots, t_n) \rrbracket = \emptyset$ .

*Proof.* If  $\exists i \in [1, n], \llbracket t_i \rrbracket = \emptyset$ , then  $\llbracket c(t_1, \dots, t_n) \rrbracket = \emptyset$ . Hence,  $\llbracket c(t_1, \dots, t_n) \rrbracket = \emptyset$ .

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

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

First we analyse the deep semantics  $\llbracket x_s^{-p} \setminus r \rrbracket$ . We can observe that  $\llbracket x_s^{-p} \setminus r \rrbracket \subseteq \llbracket x_s \setminus (r+p) \rrbracket$ , 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  $\llbracket c(p_1, \dots, p_n) \setminus c'(q_1, \dots, q_n) \rrbracket = \llbracket \sum_{i=1}^n c(p_1, \dots, p_i \setminus q_i, \dots, p_n) \rrbracket$  and  $\llbracket c(p_1, \dots, p_n) \setminus c'(q_1, \dots, q_m) \rrbracket = \llbracket c(p_1, \dots, p_n) \rrbracket$ . Therefore, given  $c, r$  and  $p$  we can construct a set denoted  $\mathcal{Q}_c(r+p)$  of  $n$ -tuples  $q = (q_1, \dots, q_n)$ , with  $n$  the arity of  $c$ , by successively distributing the patterns of  $r+p$  that have the given constructor  $c$  as head (denoted  $r^1, \dots, r^k$ ), as follows:

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

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

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

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

Given  $\mathcal{Q}_c(u)$  and  $\mathcal{Q}'_c(u)$  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  $r = \perp \vee r : s$ , and  $S$  a finite set of such couples,

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

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

$$\bullet \llbracket x_s^{-p} \setminus r \rrbracket_S = \begin{cases} \emptyset & \text{if } \llbracket x_s \setminus (r+p) \rrbracket = \emptyset \\ S & \text{else if } \exists (s, r') \in S, \llbracket r' \rrbracket = \llbracket r \rrbracket \\ \emptyset & \text{else if } \forall c \in \mathcal{C}_s, \mathcal{Q}_c^{\text{S} \cup \{(s, r)\}}(r+p) = \emptyset \\ \{(s, r)\} \cup \bigcup_{c \in \mathcal{C}_s} \bigcup_{q \in \mathcal{Q}_c^{\text{S} \cup \{(s, r)\}}(r+p)} \bigcup_{i \in [1, n]} \llbracket x_{s_i}^{-p} \setminus q_i \rrbracket_{\text{S} \cup \{(s, r)\}} & \text{else} \end{cases}$$

with  $\mathcal{Q}_c^S(u) = \{q \mid q \in \mathcal{Q}_c(u) \wedge \forall i \in [1, n], \llbracket x_{s_i}^{-p} \setminus r \rrbracket_S \neq \emptyset\}$

Moreover, we note  $\llbracket x_s^{-p} \setminus r \rrbracket_S = \bigcup_{(s', p') \in (\llbracket x_s^{-p} \setminus r \rrbracket_S)_S} \llbracket x_{s'}^{-p} \setminus p' \rrbracket$



**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  $[x_s^{-p} \setminus r] \neq \emptyset$  if and only if  $[x_s^{-p} \setminus r]_\emptyset \neq \emptyset$*

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

As  $\llbracket r \rrbracket = \llbracket r' \rrbracket$  implies  $\llbracket x_s^{-p} \setminus r \rrbracket = \llbracket x_s^{-p} \setminus r' \rrbracket$ , by construction, if  $\llbracket x_s^{-p} \setminus r \rrbracket \neq \emptyset$ , then there exists such a tree. And conversely, if there exists such a tree, as for all nodes  $\langle (s', p'), [\dots] \rangle, \llbracket x_{s'} \setminus (p' + p) \rrbracket \neq \emptyset$ , 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  $t \in \llbracket x_s^{-p} \setminus r \rrbracket$ , and thus  $\llbracket x_s^{-p} \setminus r \rrbracket \neq \emptyset$ .

We prove now that  $[x_s^{-p} \setminus r]_\emptyset \neq \emptyset$  if and only if there exists such a tree. If there exists such tree, we can prove that for each node  $N = \langle (s', p'), [\dots] \rangle$  of this tree  $[x_{s'}^{-p} \setminus p']_S \neq \emptyset$  with  $S$  the set of pairs  $(\zeta, \rho)$  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  $\llbracket u + u \rrbracket = \llbracket u \rrbracket$ ). Hence, for each infinite branch, there is a node  $N = \langle (\zeta, \rho), [\dots] \rangle$  such that the path from the root of the tree to  $N$  contains a node  $\langle (\zeta, \rho'), [\dots] \rangle$  with  $\llbracket \rho \rrbracket = \llbracket \rho' \rrbracket$ , hence  $[x_{s'}^{-p} \setminus p']_S$ . We can then prove by induction that this holds for each node. Thus, we have, for the root node,  $[x_s^{-p} \setminus r]_\emptyset \neq \emptyset$ . If  $[x_s^{-p} \setminus r]_\emptyset \neq \emptyset$ , by construction of  $[x_s^{-p} \setminus r]_\emptyset$ , we can build a tree such that for each node  $N = \langle (s', p'), [\dots] \rangle$  of this tree  $[x_{s'}^{-p} \setminus p']_S \neq \emptyset$  with  $S$  the set of pairs  $(\zeta, \rho)$  of each node in the path from the root of the tree to  $N$ , with each branch of the tree terminating on a node  $\langle (\zeta, \rho), c, [\dots] \rangle$  such that  $c$  is of arity 0 or there exists a node  $\langle (\zeta, \rho'), [\dots] \rangle$  with  $\llbracket \rho \rrbracket = \llbracket \rho' \rrbracket$  in which case we can repeat infinitely the path between the 2 nodes to get the desired tree.

Thus we have  $[x_s^{-p} \setminus r] \neq \emptyset$  if and only if  $[x_s^{-p} \setminus r]_\emptyset \neq \emptyset$ .  $\square$

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

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

The base case is for  $S$  and  $r$  such that  $\exists (s, r') \in S$  with  $\llbracket r \rrbracket = \llbracket r' \rrbracket$ , in this case  $[x_s^{-p} \setminus r]_S = S$ , and thus all 4 properties hold. Let now  $S$  be a set such that  $\forall (s', r') \in S, s \neq s' \vee \llbracket r \rrbracket \neq \llbracket r' \rrbracket$ , we suppose that  $\forall c \in \mathcal{C}, \forall q \in \mathcal{Q}_c(r + p)$ , 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  $r \in P_S(r)$  but  $r \notin P_{S \cup \{(s, r)\}}(q_i)$ , hence  $P_{S \cup \{(s, r)\}}(q_i) \subset P_S(r)$ .

1. *If  $[x_s^{-p} \setminus r]_S \neq \emptyset$  then  $\forall (s', p')$  with  $q : s$  or  $q = \perp$ ,  $[x_s^{-p} \setminus r]_{S \cup \{(s', p')\}} \neq \emptyset$ ;*

*If  $s' = s$  and  $\llbracket p' \rrbracket = \llbracket r \rrbracket$ , then the property is obviously true. Otherwise, as  $[x_s^{-p} \setminus r]_S \neq \emptyset$ , we know*

that  $\llbracket x_s \setminus (r+p) \rrbracket \neq \emptyset$  and that  $\exists c \in \mathcal{C}_s$  such that  $Q_c^{\text{S}\cup\{(s,r)\}}(r+p) \neq \emptyset$ , i.e.  $\exists q \in Q_c(r+p)$  such that  $\forall i \in [1, n]$ ,  $\llbracket x_{s_i}^{-p} \setminus q_i \rrbracket_{\text{S}\cup\{(s,r)\}} \neq \emptyset$ . We can then apply the inductive property for  $\text{S}\cup\{(s,r)\}$ , hence  $\forall i \in [1, n]$ ,  $\llbracket x_{s_i}^{-p} \setminus q_i \rrbracket_{\text{S}\cup\{(s,r)\}\cup\{(s',p')\}} \neq \emptyset$ , and thus  $\llbracket x_s^{-p} \setminus r \rrbracket_{\text{S}\cup\{(s',p')\}} \neq \emptyset$ .

2.  $\forall (s', p')$  with  $p : s'$  or  $p = \perp$ , if  $\llbracket x_s^{-p} \setminus r \rrbracket_{\text{S}\cup\{(s',p')\}} \neq \emptyset$  then  $\forall S'$  such that  $\llbracket x_{s'}^{-p} \setminus p' \rrbracket_{S'} \neq \emptyset$ ,  $\llbracket x_s^{-p} \setminus r \rrbracket_{\text{S}\cup S'} \neq \emptyset$ ;

We proceed the exact same way.

3.  $\forall (s', p')$ , with  $p : s'$  or  $p = \perp$ , if  $\llbracket x_{s'}^{-p} \setminus p' \rrbracket_S \neq \emptyset$ , then  $\llbracket x_s^{-p} \setminus r \rrbracket_{\text{S}\cup\{(s',p')\}} \subseteq \llbracket x_s^{-p} \setminus r \rrbracket_S$ ;

If  $s' = s$  and  $\llbracket p' \rrbracket = \llbracket r \rrbracket$ , then the property is obviously true. Otherwise, as  $\llbracket x_{s'}^{-p} \setminus p' \rrbracket_S \neq \emptyset$ , we have, thanks to the previous 2 properties,  $\forall (s'', p'')$ ,  $\llbracket x_{s''}^{-p} \setminus p'' \rrbracket_{\text{S}\cup\{(s,r)\}} \neq \emptyset$  if and only if  $\llbracket x_{s''}^{-p} \setminus p'' \rrbracket_{\text{S}\cup\{(s,r),(s',p')\}} \neq \emptyset$ . Thus, for all  $c \in \mathcal{C}_s$ ,  $Q_c^{\text{S}\cup\{(s',p'),(s,r)\}}(r+p) = Q_c^{\text{S}\cup\{(s',p')\}}(r+p)$ . Therefore, if one is empty, so is the other, and both semantics then verify the property. Finally, if neither  $\llbracket x_s \setminus (r+p) \rrbracket$  nor all  $Q_c^{\text{S}}(r+p)$  are empty, we have:

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

And so  $\llbracket x_s^{-p} \setminus r \rrbracket_{\text{S}\cup\{(s',p')\}} \subseteq \llbracket x_s^{-p} \setminus r \rrbracket_S$ .

4.  $\forall (s', p')$ , with  $p : s'$  or  $p = \perp$ , if  $\llbracket x_{s'}^{-p} \setminus p' \rrbracket_S \neq \emptyset$ , then  $\llbracket x_s^{-p} \setminus r \rrbracket_S \subseteq \llbracket x_s^{-p} \setminus r \rrbracket_{\text{S}\cup\{(s',p')\}} \cup \llbracket x_{s'}^{-p} \setminus p' \rrbracket_S$ ;

We proceed the exact same way. □

**Proposition 4.3.** *Given  $s \in \mathcal{S}$ ,  $p \in \mathcal{T}_\perp(\mathcal{C}, \mathcal{X})$  and  $r : s$  a sum of constructor patterns, we have*

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

with  $R = \text{getReachable}(s, p, \emptyset, r)$ . Moreover, we have  $\llbracket x_s^{-p} \setminus r \rrbracket = \emptyset$  iff  $R = \emptyset$ .

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

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

and:

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

Therefore, we then have:

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

And thus, for  $S = \emptyset$  and since  $\llbracket x_s^{-p} \setminus r \rrbracket_{\emptyset} = \emptyset \iff \llbracket x_s^{-p} \setminus r \rrbracket = \emptyset$ :

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

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

Finally, by looking at the algorithm, we can observe that  $\text{getReachable}(s, p, S, r) = \llbracket x_s^{-p} \setminus r \rrbracket_S$ . 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  $\llbracket x_s \setminus r \rrbracket = \emptyset$ . And the third *if* leads to **returning**  $S$  on (R2) when  $\exists (s, r') \in S, \llbracket r' \rrbracket = \llbracket r \rrbracket$ .

If the algorithm did not **return** on (R1) or (R2), we then have, with the conflated  $r$ ,  $\llbracket x_s^{-p} \setminus r \rrbracket = \llbracket \sum_{c \in \mathcal{C}_s} c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \setminus r \rrbracket$ . 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  $\text{getReachable}$  on the couples  $(s_i, q_i)$  obtained this way, and updates  $R$  with the results obtained.

Moreover, by considering  $P_A(u) = \{t \mid t \in P(u) \wedge \forall (s', p') \in A, \llbracket t \rrbracket \neq \llbracket p' \rrbracket\}$ , 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  $\forall c \in \mathcal{C}_s, q \in Q_c(r), i < \text{arity}(c)$  we have  $P_{\setminus(S \cup \{(s, r)\})}(q_i) \subsetneq P_S(r)$ , 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  $\llbracket x_s^{-p} \setminus r \rrbracket = \emptyset$  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,  $\llbracket c(x_{s_1}^{-p} \setminus q_1, \dots, x_{s_n}^{-p} \setminus q_n) \rrbracket$  is empty if and only if  $\exists i$  such that  $\llbracket x_{s_i}^{-p} \setminus q_i \rrbracket = \emptyset$ , 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  $\llbracket x_s^{-p} \setminus r \rrbracket_S$  on **return** (R3). □

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

*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  $\llbracket \delta(p_1, \dots, p_n) \rrbracket = \{\delta(t_1, \dots, t_n) \mid (t_1, \dots, t_n) \in \llbracket p_1 \rrbracket \times \dots \times \llbracket p_n \rrbracket\}$  and the ground semantics of  $\perp$  is empty, so is the semantics of  $\delta(v_1, \dots, \perp, \dots, v_n)$ . Hence the equality of ground semantics of the 2 sides of the rule. For the rule (S1), we have:

$$\begin{aligned} \llbracket \delta(v_1, \dots, v_i + w_i, \dots, v_n) \rrbracket &= \{\delta(t_1, \dots, t_n) \mid (t_1, \dots, t_n) \in \llbracket v_1 \rrbracket \times \dots \times \llbracket v_i + w_i \rrbracket \times \dots \times \llbracket v_n \rrbracket\} \\ &= \{\delta(t_1, \dots, t_n) \mid (t_1, \dots, t_n) \in \llbracket v_1 \rrbracket \times \dots \times \llbracket v_i \rrbracket \cup \llbracket w_i \rrbracket \times \dots \times \llbracket v_n \rrbracket\} \\ &= \{\delta(t_1, \dots, t_n) \mid (t_1, \dots, t_n) \in \llbracket v_1 \rrbracket \times \dots \times \llbracket v_i \rrbracket \times \dots \times \llbracket v_n \rrbracket\} \\ &\quad \cup \{\delta(t_1, \dots, t_n) \mid (t_1, \dots, t_n) \in \llbracket v_1 \rrbracket \times \dots \times \llbracket w_i \rrbracket \times \dots \times \llbracket v_n \rrbracket\} \\ &= \llbracket \delta(v_1, \dots, v_i, \dots, v_n) \rrbracket \cup \llbracket \delta(v_1, \dots, w_i, \dots, v_n) \rrbracket \end{aligned}$$

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

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

**(T3):** If  $v \in \llbracket \alpha(v_1, \dots, v_n) \times \alpha(w_1, \dots, w_n) \rrbracket$ , then  $v \in \llbracket \alpha(v_1, \dots, v_n) \rrbracket$  and  $v \in \llbracket \alpha(w_1, \dots, w_n) \rrbracket$ . As  $\llbracket \alpha(p_1, \dots, p_n) \rrbracket = \{ \alpha(t_1, \dots, t_n) \mid (t_1, \dots, t_n) \in \llbracket p_1 \rrbracket \times \dots \times \llbracket p_n \rrbracket \}$ ,  $v = \alpha(t_1, \dots, t_n)$  such that  $\forall i \in [1, n], t_i \in \llbracket v_i \rrbracket$  and  $t_i \in \llbracket w_i \rrbracket$ . Therefore,  $\forall i \in [1, n], t_i \in \llbracket v_i \rrbracket \cap \llbracket w_i \rrbracket$  and thus  $v \in \llbracket \alpha(v_1 \times w_1, \dots, v_n \times w_n) \rrbracket$ . Hence the first inclusion. Similarly, we can show that if  $v \in \llbracket \alpha(v_1 \times w_1, \dots, v_n \times w_n) \rrbracket$ , then  $v \in \llbracket \alpha(v_1, \dots, v_n) \times \alpha(w_1, \dots, w_n) \rrbracket$ , 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  $\llbracket v \rrbracket \subseteq \llbracket x_s^{-1} \rrbracket$ . Hence the equality of ground semantics of the 2 sides of these rules.

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

$$\llbracket x_s^{-p} \rrbracket = \bigcup_{c \in \mathcal{C}_s} \llbracket c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \setminus p \rrbracket \quad (1)$$

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

Therefore, we have  $\llbracket x_s^{-p} \rrbracket = \llbracket \sum_{c \in \mathcal{C}_s} c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \setminus p \rrbracket$  hence

$$\begin{aligned} \llbracket x_s^{-p} \times \alpha(v_1, \dots, v_n) \rrbracket &= \llbracket \left( \sum_{c \in \mathcal{C}_s} c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \setminus p \right) \times \alpha(v_1, \dots, v_n) \rrbracket \\ &= \left( \bigcup_{c \in \mathcal{C}_s} \llbracket c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \setminus p \rrbracket \right) \cap \llbracket \alpha(v_1, \dots, v_n) \rrbracket \\ &= \left( \bigcup_{c \in \mathcal{C}_s} \llbracket c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \rrbracket \setminus \llbracket p \rrbracket \right) \cap \llbracket \alpha(v_1, \dots, v_n) \rrbracket \\ &= \bigcup_{c \in \mathcal{C}_s} \llbracket c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \rrbracket \cap (\llbracket \alpha(v_1, \dots, v_n) \rrbracket \setminus \llbracket p \rrbracket) \\ &= \llbracket \sum_{c \in \mathcal{C}_s} c(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \times (\alpha(v_1, \dots, v_n) \setminus p) \rrbracket \end{aligned}$$

□

**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*

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

*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, for all extended pattern  $t$  and  $u$ ,  $\forall \omega \in \mathcal{Pos}(t)$ ,  $t[u]_\omega$  *regular* implies  $u$  *regular* and for all *regular* extended  $v$ ,  $t[v]_\omega$  is *regular*. Thus all rules preserve the *regular* property.  $\square$

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

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

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

*Moreover the normal form  $t \downarrow_{\mathcal{R}_p} = v$  is either  $\perp$  or a pure term such that  $\forall \omega_+ \in \mathcal{Pos}(v)$ ,  $v(\omega_+) = + \implies \forall \omega < \omega_+$ ,  $v(\omega) = +$  and  $\llbracket t \rrbracket = \emptyset$  if and only if  $v = \perp$ .*

*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  $c(v_1, \dots, v_i + u_i, \dots, v_n)$ , 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  $t \downarrow_{\mathcal{R}_p} = \perp$  then Proposition 4.4 ensures that  $\llbracket t \rrbracket = \llbracket \perp \rrbracket = \emptyset$ . We note  $v = t \downarrow_{\mathcal{R}_p}$ , once again we know that  $\llbracket t \rrbracket = \llbracket v \rrbracket$ , let's prove that if  $\llbracket v \rrbracket = \emptyset$  then  $v = \perp$ . We suppose that  $v \neq \perp$  and we prove by induction that  $v$  is not in normal form.

If  $v = x_s^{-P} \setminus u$  and  $\llbracket v \rrbracket = \emptyset$ , then rule P6 applies, thus  $v$  is not in normal form. If  $v = v_1 + v_2$ , then  $\llbracket v_1 \rrbracket = \llbracket v_2 \rrbracket = \emptyset$ , thus by induction, either  $v_1 = v_2 = \perp$  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  $v = c(v_1, \dots, v_n)$ , then  $\exists i \in [1, n]$  such that  $\llbracket v_i \rrbracket = \emptyset$ , thus by induction, either  $v_i = \perp$  and rule E1 applies or  $v_i$  is not in normal form. In both cases,  $v$  is not in normal form.

Therefore, if  $v \neq \perp$  is a quasi-additive such that  $\llbracket v \rrbracket = \emptyset$ , then  $v$  is not in normal form. Thus, if  $\llbracket t \rrbracket = \emptyset$ , then  $t \downarrow_{\mathcal{R}_p} = \perp$ .  $\square$

**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  $t = c(t_1, \dots, t_n)$ , 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  $v = (v_1 + v_2) \downarrow_{\mathcal{R}_p} = (t \setminus u) \downarrow_{\mathcal{R}_p}$  is a quasi-additive pattern such that  $depth(v) \leq depth(v_1 + v_2) \leq depth(t \setminus u)$ .
- if  $u = c(u_1, \dots, u_n)$ , 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  $t = c'(t_1, \dots, t_m)$ , 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  $\sum_{i \in [1, m]} c(t_1, \dots, t_i \setminus u_i, \dots, t_n)$ , 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,  $t \setminus u \twoheadrightarrow_{\mathcal{R}_p} \sum_{i \in [1, m]} c(t_1, \dots, v_i, \dots, t_n)$  and by monotonicity,  $depth(\sum_{i \in [1, m]} c(t_1, \dots, v_i, \dots, t_n)) \leq depth(t \setminus u)$ . Moreover,  $w = \sum_{i \in [1, m]} c(t_1, \dots, v_i, \dots, t_n)$  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 confluence,  $v = (t \setminus u) \downarrow_{\mathcal{R}_p}$  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  $t = c(t_1, \dots, t_n)$  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  $t = c'(t_1, \dots, t_n)$  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  $\sum_{d \in \mathcal{C}_s} d(x_{s_1}^{-p}, \dots, x_{s_n}^{-p}) \times (c() \setminus p)$ . 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  $p = c'(p_1, \dots, p_n)$  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  $u = x_s^{-1} \setminus v$  to prove the induction step on the form of  $t$ .

Finally, if  $u = u_1 + u_2$  with  $\text{depth}(u_1) = \text{depth}(u_2) = 1$ , then rule S3 applies to  $t \setminus u$  and reduces it to  $(t \times u_1) + (t \times u_2)$ . Moreover, by induction on  $u_1$  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  $(v_1 + v_2) \downarrow_{\mathcal{R}_p} = (t \setminus u) \downarrow_{\mathcal{R}_p}$  is a quasi-additive pattern.

We now suppose  $\text{depth}(u) = n > 1$  and for all quasi-additive pattern  $v$  such that  $\text{depth}(v) < n$ , for all quasi-additive pattern  $\tau$ ,  $(\tau \setminus v) \downarrow_{\mathcal{R}_p}$  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  $u = c(u_1, \dots, u_m)$  with  $\forall i \in [1, m], \text{depth}(u_i) < n$ , we proceed by induction on the form of  $t$ . For the case when  $t = c'(t_1, \dots, t_{m'})$ , 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  $c(t_1 \times u_1, \dots, t_{m'} \times u_{m'})$ . 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  $c(v_1, \dots, v_m)$  is a quasi-additive pattern and we know, thanks to Lemma B.5, that  $c(v_1, \dots, v_m) \downarrow_{\mathcal{R}_p} = (t \setminus u) \downarrow_{\mathcal{R}_p}$  is a quasi-additive pattern. If  $t = x_s^{-p}$ , then rule T4 applies to  $t \setminus u$  and reduces it to  $\sum_{d \in \mathcal{C}_s} d(x_{s_1}^{-p}, \dots, x_{s_{m'}}^{-p}) \times (u \setminus p)$ . 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  $\perp$  or a sum of quasi-additive patterns of the form  $c(w_1, \dots, w_n)$ , with  $\forall i \in [1, m], \text{depth}(w_i) < n$ . If  $(u \setminus p) \downarrow_{\mathcal{R}_p} = \perp$ , then rule E3 applies and thus  $(t \times u) \downarrow_{\mathcal{R}_p} = \perp$ . Otherwise, by applying recursively S2/S3, T4/T3, and A1/A2 we get  $t \times u \twoheadrightarrow_{\mathcal{R}_p} \sum c(x_{s_1}^{-p} \times w_1, \dots, x_{s_m}^{-p} \times w_m)$ . 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  $\sum c(v_1, \dots, v_m)$  is a quasi-additive pattern and we know, thanks to Lemma B.5, that  $(\sum c(v_1, \dots, v_m)) \downarrow_{\mathcal{R}_p} = (t \setminus u) \downarrow_{\mathcal{R}_p}$  is a quasi-additive pattern. Finally, if  $t = t_1 + t_2$ , we proceed exactly the same way as in the case when  $u = x_s^{-1} \setminus v$  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  $\text{depth}(u) = 1$  to prove the induction step on the form  $u$  when  $\text{depth}(u) > 1$ .  $\square$

**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  $t \times p \twoheadrightarrow_{\mathcal{R}_p} \perp$  if and only if  $\llbracket t \times p \rrbracket = \emptyset$ .*

*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  $\perp$ .  $\square$



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

The meta encoding of the rule schemas in Figure 1 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
)

```