Pattern eliminating transformations

Pierre Lermusiaux, Horatiu Cirstea, Pierre-Etienne Moreau

Pattern eliminating transformations

Pierre Lermusiaux, Horatiu Cirstea, and Pierre-Etienne Moreau

Université de Lorraine – LORIA
name.surname@loria.fr

Abstract

Program transformation is a common practice in computer science, and its many applications can have a range of different objectives. For example, a program written in an original high level language could be either translated into machine code for execution purposes, or towards a language suitable for formal verification. Languages often have a lot of different constructions, and thus, transformations often focus on eliminating some of these constructions, or at least processing some specific sequence of constructions. Rewriting is a widely established formalism to describe the mechanism and the logic behind such transformations. It relies mainly on the principle of rewrite rules to describe the different operations performed. Generally type-preserving, these rewrite rules can ensure that the transformation result has a given type and thus give syntactic guarantees on the constructions it consists of. However, we sometimes want the transformation to provide more guarantees on the shape of its result by specifying that some patterns of constructions does not appear. For this purpose, we propose in this paper an approach based on annotating transformation function symbols with (anti-)pattern in order for such transformation to guarantee stronger properties on the shape of its result. With the generic principles governing term algebra and rewriting, we believe this approach to be an accurate formalism to any language providing pattern-matching primitives.

1 Introduction

Rewriting is a widely established formalism for a number of applications in both computer science and mathematics and has been used, in particular, to describe program semantics [13] and transformations [11, 4]. In general, compilation consists in several phases, also called passes, eventually obtaining a program in a different target language. These phases use some corresponding intermediate languages which generally contain less and less constructions of the original language.

Consider, for example, a very simple language allowing to express some form of λ-expressions:

$$\begin{array}{rcccc} Expr & = & Var(String) & | & Apply(Expr, Expr) \\ & | & Lambda(String, Expr) & | & Let(String, Expr, Expr) \end{array}$$

A first step in the compilation of such expressions is to eliminate the *Let* constructor using the rewrite rule $Let(name, e_1, e_2) \twoheadrightarrow Apply(Lambda(name, e_2), e_1)$ to obtain pure λ-expressions.

The goal is to provide a formalism allowing to describe such transformations and to guarantee that some language constructs are eliminated by these transformations.

Existing works addressed this problem from different perspectives. In particular, a number of approaches rely on the use of automata to implement transformation. Such approach is particularly relevent as the input and output of the transformation can be viewed as a regular language [1] or a tree [3]. While this approach focus on the decidability and complexity of some specific cases it does not handle the general problems we target here. Functionnal approaches to transformation [12] focus on the simplicity and expressiveness of the formalism more than on verifying the elimination of constructs. This verification can be performed by using fine grained typing systems which combine overloading, subtyping and polymorphism through the use of variants [7] but is somewhat limited w.r.t. the type of constructs which can be eliminated.

We use rewriting to define such transformations and annotate the function symbols in order to check that the associated transformation verifies the desired elimination properties.

First, in the next section, we will introduce all the notions and notations used in the article. We will then, in the following section, give a formal definition to the property we want pattern eliminating transformation to guarantee: pattern-freeness. The 2 subsequent sections will be focused respectively on introducing a method to reliably verify such properties and on studying how they can be guaranteed by the considered transformations, using and extending notions of pattern semantics introduced in the notions of the next section.

2 Pattern matching and term rewriting systems

We define in this section the basic notions and notations used in this paper; more details can be found in [2, 14].

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 constructors symbols from functions 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* $\mathcal{D}om(f) = s_1 \times \ldots \times s_n \in \mathcal{S}^*$ and *co-domain* $\mathcal{C}o\mathcal{D}om(f) = s \in \mathcal{S}$ is written $f : s_1 \times \ldots \times s_n \mapsto s$. Variables are also sorted and we write $x : s$ or x^s to indicate that variable x has sort s. The set $\mathcal{X}_s$ denotes a set of variables of sort s and $\mathcal{X} = \bigcup_{s \in \mathcal{S}} \mathcal{X}_s$ is the set of sorted variables.

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

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

We call *substitution* any mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ which is the identity except over a finite set of variables called its domain; any substitution extends as expected to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Given a sort s, a value $v : s$ and a constructor pattern p, we say that p *matches* v (denoted $p \lessdot v$) if there exists a substitution σ such that $v = \sigma(p)$.

A *constructor rewrite rule* is of the form $\varphi(l_1, \ldots, l_n) \twoheadrightarrow r \in \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \times \mathcal{T}_s(\mathcal{F}, \mathcal{X})$ with $s \in \mathcal{S}$, $\varphi \in \mathcal{D}$, $l_1, \ldots, l_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ and such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A *constructor term rewriting system* (CTRS) is a set of rewrite rules $\mathcal{R}$ inducing a *rewriting relation* over $\mathcal{T}(\mathcal{F})$, denoted by $\longrightarrow_{\mathcal{R}}$ and such that $t \longrightarrow_{\mathcal{R}} t'$ iff there exist $l \twoheadrightarrow r \in \mathcal{R}$, $\omega \in \mathcal{P}os(t)$, and a substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t[\sigma(r)]_\omega$.

Starting from the observation that a pattern can be interpreted as the set of its instances, the notion of *ground semantics* was introduced in [6] as the set of all ground constructor instances of a pattern $p \in \mathcal{T}_s(\mathcal{C}, \mathcal{X})$: $[\![p]\!] = \{\sigma(p) \mid \sigma(p) \in \mathcal{T}_s(\mathcal{C})\}$.

Proposition 1. *Given a pattern p and a value v, $v \in [\![p]\!]$ iff $v = \sigma(p)$.*

Note that the ground semantics of a variable x^s is the set of all possible ground patterns: $[\![x^s]\!] = \mathcal{T}_s(\mathcal{C})$, and since patterns are linear we can use a recursive definition for the non variable patterns: $[\![c(p_1, \ldots, p_n)]\!] = \{c(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in [\![p_1]\!] \times \ldots \times [\![p_n]\!]\}, \forall c \in \mathcal{C}$. Moreover

$[\![x^s]\!] = \{c(t_1,\ldots,t_n) \mid c \in \mathcal{C} \text{ s.t. } c : s_1 \times \ldots \times s_n \mapsto s \wedge \forall i, t_i \in [\![x^{s_i}]\!]\} = \bigcup_{c \in \mathcal{C}_s} [\![c(x_1,\ldots,x_n)]\!]$.

We consider also a special pattern $\bot$ with $[\![\bot]\!] = \emptyset$. Given two patterns p, q we can compute [6] their complement $p \setminus q$, *i.e.* a set $p_1,\ldots,p_n$ of patterns s.t. $[\![p \setminus q]\!] = [\![p_1,\ldots,p_n]\!]$.

3 Pattern free terms

The sort provides some information on the shape of the terms of the respective sort and in particular allows one to check if a given symbol may be present or not in these terms. In fact, the precise language of the values of a given sort is implicitly given by the signature. Sorts are less informative concerning the shape of the values obtained when reducing terms containing defined symbols since they strongly depend on the CTRS defining these symbols.

We want to ensure that the normal form of a term, if it exists, does not contain a constructor and more generally that no subterm of this normal form matches a given pattern. For this we annotate all defined symbols with the patterns that are supposed to be absent from the normal form and we check that the CTRS defining the corresponding functions are consistent with these annotations. We will see later on how this consistence can be verified and we focus first on the notion of pattern-free term and the corresponding ground semantics.

We consider that every defined symbol $f \in \mathcal{D}$ is annotated with a pattern $p \in \mathcal{T}(\mathcal{C},\mathcal{X}) \cup \{\bot\}$ and we use this notation to define *pattern-free* terms. Intuitively, a term of the form $f^{-p}(t_1,\ldots,t_n)$ should ultimately be reduced to a value containing no subterms matched by p.

Definition 3.1 (Pattern-free)**.** *Given a pattern $p \in \mathcal{T}(\mathcal{C},\mathcal{X}) \cup \{\bot\}$,*

- *a value $v \in \mathcal{T}(\mathcal{C})$ is p-free iff $\forall \omega \in \mathcal{P}os(v), p \not\lessdot v_{|\omega}$;*
- *a ground term $t \in \mathcal{T}(\mathcal{F})$ is p-free iff $\forall \omega \in \mathcal{P}os(t)$ such that $t_{|\omega} = f^{-q}(t_1,\ldots,t_n)$ with $f \in \mathcal{D}, q \in \mathcal{T}(\mathcal{C},\mathcal{X}) \cup \{\bot\}$, $\mathcal{C}o\mathcal{D}om(f) = s$, we have $\forall v \in \mathcal{T}_s(\mathcal{C})$ q-free, $t[v]_\omega$ is p-free;*
- *a linear term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is p-free iff $\forall \sigma$ such that $\sigma(t) \in \mathcal{T}(\mathcal{F})$, $\sigma(t)$ is p-free.*

A value is p-free if and only if p matches no subterm of the value. A ground term is p-free if and only if replacing (all) the subterms headed by a defined symbol f^{-q} by any appropriate q-free value results in a p-free term. For general terms verifying the pattern-freeness comes to verifying the property for all ground instances of the term. While the pattern-freeness of a value can be checked by exploring all its subterms this is not possible for a general term since we potentially have to check the pattern-freeness of an infinite number of values. We present in the next section an approach for solving this problem.

4 Extensions of ground semantics

The previously introduced ground semantics can be used to compare the shape of the root of a constructor pattern p_1 to another constructor pattern p_2. Indeed, by definition, if $[\![p_1]\!] \cap [\![p_2]\!] = \emptyset$, then $\forall \sigma, \sigma(p_1) \notin [\![p_2]\!]$ so $p_2 \not\lessdot \sigma(p_1)$.

Example 4.1. *Consider the signature Σ with $\mathcal{S} = \{s_1, s_2\}$ and $\mathcal{F} = \mathcal{C} = \{c_1 : s_2, s_1 \mapsto s_1, c_2 : s_2 \mapsto s_1, c_3 : [] \mapsto s_1, c_4 : s_2 \mapsto s_2, c_5 : s_1 \mapsto s_2\}$. Using the transformation method proposed in [6] the complement $c4(c3()) \setminus c4(x)$ is reduced to $\bot$ indicating that $[\![c4(c3()) \setminus c4(x)]\!] = [\![c4(c3())]\!] \setminus [\![c4(x)]\!] = \emptyset$ and thus that $c4(c3())$ is redundant w.r.t. $c4(x)$. Moreover $c4(c3()) \setminus (c4(c3()) \setminus c4(x))$ is reduced to $c4(c3()) \setminus \bot$ and then to $c4(c3())$ indicating now that $[\![c4(c3())]\!] \cap [\![c4(x)]\!] \neq \emptyset$ and thus that $c4(c3())$ is not $c4(x)$ free.*

Similarly, $c4(c3()) \setminus (c4(c3()) \setminus c5(x))$ and $c3() \setminus (c3() \setminus c5(x))$ are both reduced to $\bot$ and consequently we can deduce that $c4(c3())$ is $c5(x)$-free.

The pattern-freeness properties in the above example could have been checked by trying to match all the subterms and we can see this method as a starting point for the generalisation to general terms. We first introduce an extended ground semantics:

Definition 4.1 (Extended ground semantics). *Given $s \in \mathcal{S}$ and a pattern $p \in \mathcal{T}(\mathcal{C},\mathcal{X}) \cup \{\bot\}$*

- $[\![x^s_{-p}]\!] = \{v \mid v \in \mathcal{T}_s(\mathcal{C}) \wedge v \text{ p-free}\}$;
- $[\![f^{-p}(t_1,\ldots,t_n)]\!] = [\![x^s_{-p}]\!]$ *with* $\mathcal{CoDom}(f) = s$;
- $[\![c(t_1,\ldots,t_n)]\!] = \{c(v_1,\ldots,v_n) \mid v_i \in [\![t_i]\!]\}$ *with* $c : s_1 \times \ldots \times s_n \mapsto s \in \mathcal{C}, t_i \in \mathcal{T}_{s_i}(\mathcal{F},\mathcal{X})$.

The 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-freeness, this by taking into account the annotation of the respective defined symbol. When restricting to patterns we retrieve the original definition of ground semantics.

The extended ground semantics of a p-free variable of sort s can be also defined as:

$$[\![x^s_{-p}]\!] \quad = \quad \bigcup_{c \in \mathcal{C}_s} [\![c(x^{s_1}_{-p},\ldots,x^{s_i}_{-p})]\!] \setminus [\![p]\!]$$

and when $p = \bot$ we retrieve the corresponding definition for the classical ground semantics. This observation allows us to easily adapt the method introduced in [6] for computing constructor pattern complements to general annotated terms.

We can now establish pattern-free properties using this extended ground semantics and the *reachable sorts w.r.t.* a given sort s: $\lfloor s \rfloor = \{s' \mid \exists\, t \in \mathcal{T}_s(\mathcal{C}), \omega \in \mathcal{P}os(t) \text{ s.t. } t_{|\omega} : s'\}$

Proposition 2 (Pattern-free vs Extended Ground Semantics). *Let $p \in \mathcal{T}(\mathcal{C},\mathcal{X}), t \in \mathcal{T}_s(\mathcal{F},\mathcal{X})$:*

- *if $t = x^s$ and $\forall s' \in \lfloor s \rfloor, [\![x^{s'}_{\bot}]\!] \cap [\![p]\!] = \emptyset$ then t is p-free;*
- *if $t = f^{-q}(t_1,\ldots,t_n)$ and $\forall s' \in \lfloor s \rfloor, [\![x^{s'}_{-q}]\!] \cap [\![p]\!] = \emptyset$ then t is p-free;*
- *If $t = c(t_1,\ldots,t_n)$ with $c \in \mathcal{C}$, t is p-free iff $[\![t]\!] \cap [\![p]\!] = \emptyset$ and $\forall i \in [1,n]$, t_i is p-free.*

The extended semantics of annotated terms together with the corresponding transformation of complement patterns (corresponding to semantics intersections) into sets of constructor patterns over-approximating their semantics allows us to systematically check if a term is pattern-free. Unfortunately equivalent extended semantics do not guarantee the preservation of pattern-freeness: having $[\![u]\!] = [\![v]\!]$ and u p-free does not necessarily mean that v is p-free.

We introduce the notion of *deep semantics* of a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, denoted $\{\!|t|\!\}$, which can be seen as an over-approximation of the set of all subterms of all values of its ground semantics, *i.e.* $\{u_{|\omega} \mid u \in [\![t]\!], \omega \in \mathcal{P}os(u)\} \subseteq \{\!|t|\!\}$:

Definition 4.2. *Given the sorts $s_1,\ldots,s_n,s \in \mathcal{S}$ and a pattern $p \in \mathcal{T}(\mathcal{C},\mathcal{X}) \cup \{\bot\}$*

- $\{\!|x^s_{-p}|\!\} = \bigcup_{s' \in \lfloor s \rfloor} [\![x^{s'}_{-p}]\!]$;
- $\{\!|f^{-p}(t_1,\ldots,t_n)|\!\} = \bigcup_{s' \in \lfloor s \rfloor} [\![x^{s'}_{-p}]\!]$ *with* $f : s_1 \times \ldots \times s_n \mapsto s \in \mathcal{D}$;
- $\{\!|c(t_1,\ldots,t_n)|\!\} = [\![c(t_1,\ldots,t_n)]\!] \cup \left(\bigcup_{i=1}^{n} \{\!|t_i|\!\}\right)$ *with* $c : s_1 \times \ldots \times s_n \mapsto s \in \mathcal{C}$.

The deep semantics of a variable must not only take into account its own sort, but also the sorts of all the subterms of its instances. Similarly, for terms headed by a constructor we consider the semantics of terms and of all its subterms.

We can now identify terms that have such a shape that a given pattern p does not appear in any of their ground instances. In other words, we have an alternative method for establishing pattern-free properties for linear terms in $\mathcal{T}(\mathcal{F},\mathcal{X})$:

Proposition 3 (Pattern-free vs Deep Semantics). *Let $p \in \mathcal{T}(\mathcal{C},\mathcal{X}), t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, if $\{\!|t|\!\} \cap [\![p]\!] = \emptyset$ then t is p-free.*

Example 4.2. *Let's consider the signature from Example 4.1: We add defined symbols* $\mathcal{D} = \{f : s_1 \mapsto s_1, g : s_2 \mapsto s_2\}$ *such that* f *is supposed to eliminate the* $p_1 = c_1(c_4(x), y)$ *and* g *eliminates* $p_2 = c_4(x)$*. We therefore have the following constructor TRS:*

$$\begin{aligned} f(c_1(c_4(x), y)) &\twoheadrightarrow c_1(g(x), f(y)) \\ f(c_1(c_5(x), y)) &\twoheadrightarrow c_1(c_5(f(x)), f(y)) \\ f(c_2(c_4(x))) &\twoheadrightarrow c_2(c_4(g(x))) \\ f(c_2(c_5(x))) &\twoheadrightarrow c_2(c_5(f(x))) \\ f(c_3()) &\twoheadrightarrow c_3() \end{aligned} \qquad \begin{aligned} g(c_4(x)) &\twoheadrightarrow c_5(c_1(g(x), c_3())) \\ g(c_5(c_1(x, y))) &\twoheadrightarrow c_5(c_1(g(x), g(c_5(y)))) \\ g(c_5(c_2(x))) &\twoheadrightarrow c_5(c_2(g(x))) \\ g(c_5(c_3())) &\twoheadrightarrow c_5(c_3()) \end{aligned}$$

Let's consider the term $c_1(g^{-p_2}(x), f^{-p_1}(y))$*.* $f^{-p_1}(y)$ *and* $g^{-p_2}(x)$ *have respectively the same semantics has* $y^{s_1}_{-p_1}$ *and* $x^{s_2}_{-p_2}$*. Therefore the deep semantics of the whole term is the union of the ground semantics of* $c_1(x^{s_2}_{-p_2}, y^{s_1}_{-p_1})$*,* $y^{s_1}_{-p_1}$*,* $x^{s_2}_{-p_1}$*,* $y^{s_1}_{-p_2}$ *and* $x^{s_2}_{-p_2}$*. Similarily as in Example 4.1, we can thus compare* $c_1(x^{s_2}_{-p_2}, y^{s_1}_{-p_1})$ *to* p_1 *and prove that it is* p_1*-free.*

5 Semantics preservation for CTRS

Pattern-freeness properties 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 CTRSs defining the annotated symbols are consistent with these annotations, *i.e.* verifying that the semantics is preserved by reduction.

Definition 5.1 (Semantics preservation)**.** *A constructor rewrite rule* $\varphi^{-p}(l_1, \ldots, l_n) \twoheadrightarrow r$ *is* semantics preserving *w.r.t. the extended semantics, resp. deep semantics, iff* $[\![r]\!] \subseteq [\![l]\!]$*, resp.* $\{\!\{r\}\!\} \subseteq \{\!\{l\}\!\}$*. A CTRS is semantics preserving iff all its rewrites rules are.*

Since the semantics of the left-hand side of a rewrite rule is the set of p-free values then such a rule is semantics preserving iff its right-hand side is p-free. It is easy to check that:

Proposition 4 (Semantics preservation)**.** *Given a semantics preserving CTRS* $\mathcal{R}$ *we have that then,* $\forall t, u \in \mathcal{T}(\mathcal{F})$*:* $t \longrightarrow_{\mathcal{R}} u \implies [\![u]\!] \subseteq [\![t]\!]$

Example 5.1. *Let's consider the case presented in Example 4.2:*

We want each of the rules of the constructor TRS to be pattern-free preserving, so that for all $v_1 \in \mathcal{T}_{s_1}(\mathcal{C})$*,* $f(v_1)$ *is and stays* $c_1(c_4(x), y)$*-free through every reduction step, and for all* $v_2 \in \mathcal{T}_{s_2}(\mathcal{C})$*,* $g(v_2)$ *is and stays* $c_4(x)$*-free through every reduction step.*

In terms of pattern-free preservation we have:

Proposition 5 (Pattern-free preservation)**.** *Given a semantics preserving CTRS* $\mathcal{R}$ *we have that* $\forall t, u \in \mathcal{T}(\mathcal{F}), p \in \mathcal{T}(\mathcal{C}, \mathcal{X})$*:* t *p-free* $\wedge\, t \longrightarrow_{\mathcal{R}} u \implies u$ *p-free*

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.

We suppose the existence of normal forms but the formalism does not rely on the termination of the analysed rewriting systems; if the property is not verified a final value is not obtained but the intermediate terms in the infinite reduction verify nevertheless the pattern-freeness properties *w.r.t.* the specified annotations. Moreover, different termination techniques and

tools [8, 10] on termination analysis can be used to analyse the termination of the rewriting systems we addressed in this paper.

We believe this formalism opens a lot of opportunities for further developments. In particular, this method could be extended in the context of automatic rewriting rule generation techniques, such as the one introduced in [5], in order to implement transformation approaches of passes such as in [9]. Indeed, the formalism considered here relies on the same pattern matching primitives as these techniques.

References

[1] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 599–610. ACM, 2011.

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

[3] Patrick Bahr and Laurence E. Day. Programming macro tree transducers. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, pages 61–72. ACM, 2013.

[4] Françoise Bellegarde. Program transformation and rewriting. In Ronald V. Book, editor, *Rewriting Techniques and Applications*, pages 226–239, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[5] Horatiu Cirstea, Sergueï Lenglet, and Pierre-Etienne Moreau. A faithful encoding of programmable strategies into term rewriting systems. In *RTA 2015*, volume 36 of *LIPIcs*, pages 74–88, 2015.

[6] Horatiu Cirstea and Pierre-Etienne Moreau. Generic encodings of constructor rewriting systems. arxiv:1905.06233, arXiv, 2019.

[7] Jacques Garrigue. Programming with polymorphic variants. In *In ACM Workshop on ML*, 1998.

[8] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7, 2011.

[9] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 343–350. ACM, 2013.

[10] Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle functions via termination of rewriting. In *ITP 2011*, pages 152–167, 2011.

[11] David Lacey and Oege de Moor. Imperative program transformation by rewriting. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 52–68, London, UK, UK, 2001. Springer-Verlag.

[12] François Pottier. Visitors unchained. *Proceedings of the ACM on Programming Languages*, 1(ICFP):28:1–28:28, 2017.

[13] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

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