# Generic Encodings and Static Analysis of Constructor Rewriting Systems

Horatiu Cirstea
University of Lorraine, CNRS, Inria, LORIA
Nancy, France
Horatiu.Cirstea@loria.fr

Pierre Lermusiaux
Inria, IRISA
Rennes, France
Pierre.Lermusiaux@inria.fr

Pierre-Etienne Moreau
University of Lorraine, CNRS, Inria, LORIA
Nancy, France
Pierre-Etienne.Moreau@loria.fr

***Abstract*—Rewriting is a formalism widely used in computer science and mathematical logic. The classical formalism has been extended, in the context of functional languages, with an order over the rules and, in the context of rewrite based languages, with the negation over patterns. We have proposed a concise and clear algorithm computing the difference over patterns which can be used to define generic encodings of constructor term rewriting systems with negation and order into classical term rewriting systems. As a direct consequence, established methods used for term rewriting systems can be applied to analyze properties of the extended systems. The approach can also be seen as a generic compiler which targets any language providing basic pattern matching primitives. The formalism provides also a new method for deciding if a set of patterns subsumes a given pattern and thus, for checking the completeness of a set of patterns, the presence of useless patterns, or the absence of some patterns. The latter is particularly useful when one wants to verify that some patterns are absent from the result of a particular transformation. Several extensions have been proposed to statically verify the absence of specified patterns from the reachable terms of a constructor based term rewriting systems.**

***Index Terms*—pattern matching, term rewriting, static analysis, compilation**

## I. Introduction

Rewriting is a very powerful formalism used, for example, in semantics in order to describe program behaviours [1], [2] and program transformations [3], [4], in automated reasoning when describing by inference rules a logic, a theorem prover or a constraint solver [5]. It is also used to compute in systems making the notion of rule explicit, like Mathematica [6], Maude [7], Stratego [8], or Tom [9].

Rewrite rules consist of a pattern that describes the shape of the objects to be transformed and the way they will be transformed. The application of the rewrite rules is decided locally and independently of the other rules and thus, rewriting is very practical for describing schematically and locally the transformations one wants to operate. Comparing to the general rewriting formalism, rule- and pattern-based programming languages generally use an application strategy and this is particularly convenient when an uncontrolled application of the rules might not terminate or lead to different results depending on the rule and context chosen at each step of the reduction. Simple strategies like the ordered application of rules used in functional programming languages, generally allow for concise and clear specifications which avoid an exhaustive specification of alternative and default cases.

It is interesting to analyze such programs specified using ordered rules and determine whether they terminate and what is the shape of the potential results when this is the case. On the other hand, the proof techniques and tools used are designed for uncontrolled rewriting and thus, can't be applied directly for such ordered systems. There are several works describing powerful techniques for analyzing the termination of functional programs when the rules are ordered [10]–[13] but they can't be easily adapted to be used as a compiler of ordered rules towards unrestricted rules pluggable in different languages providing pattern matching primitives.

We briefly present here the formalism introduced in [14] for computing pattern complements and which can be directly applied to transform an ordered constructor term rewriting system (CTRS) into a plain CTRS. The resulting system can then be analyzed using well-established techniques and automatic tools used for plain term rewriting system (TRS). We also describe how the approach proposed in [15], [16] can be used to verify that the results obtained when reducing a term with respect to such a CTRS have a specific shape.

## II. Generic encodings

As we have mentioned before, in functional programming languages the order rules appear in the program is significant for the evaluation. For instance [14], if we consider a term representation of motor vehicles characterised by the energy they use and their type, then we can use the following list of rules to specify some kind of eco label for each vehicle:

$$\begin{array}{l}[\, paint(car(x, suv)) \twoheadrightarrow red,\\ \;\; paint(car(electric, x)) \twoheadrightarrow blue,\\ \;\; paint(car(diesel, y)) \twoheadrightarrow red,\\ \;\; paint(car(x, y)) \twoheadrightarrow white,\\ \;\; paint(x) \twoheadrightarrow red \;\;]\end{array}$$

The system indicates that all SUVs independently of their fuel and all diesel cars independently of their type are bad for the environment (red), and that all electric cars but the SUVs (handled by the previous rule) are eco-responsible (blue). The remaining cars, not in the previous categories, are considered neutral (white). Finally, any other vehicle (which is not a car) is labeled red.

The transformation approach we have proposed [14] can be used as a generic compiler for ordered CTRS which could be easily integrated in any language providing rewrite rules, or at least pattern matching primitives. For example, if we consider trucks and cars with 4 fuel types and 3 styles this approach transforms the previous list of rules into the following order independent set of rules:

$$\begin{aligned}\{\ & paint(car(electric, sedan)) \twoheadrightarrow blue,\\ & paint(car(electric, minivan)) \twoheadrightarrow blue,\\ & paint(car(hybrid, sedan)) \twoheadrightarrow white,\\ & paint(car(hybrid, minivan)) \twoheadrightarrow white,\\ & paint(car(gas, sedan)) \twoheadrightarrow white,\\ & paint(car(gas, minivan)) \twoheadrightarrow white,\\ & paint(truck(x, y)) \twoheadrightarrow red,\\ & paint(car(x, suv)) \twoheadrightarrow red,\\ & paint(car(diesel, x)) \twoheadrightarrow red \quad \}\end{aligned}$$

Moreover, the approach can handle anti-patterns [17], i.e. patterns that may contain complement symbols. Such patterns allow the specification of negative conditions in addition to the positive conditions expressed with classical patterns. Due to their expressiveness such anti-patterns have been integrated in tools featuring pattern matching like Tom [18] and Mathematica [6].

For example, using anti-patterns we can represent all the cars that are not SUVs by the pattern $car(x, !suv)$, and all cars which are neither SUV nor diesel by the pattern $car(!diesel, !suv)$. The eco-labeling can then be expressed by the following list of rules with anti-patterns

$$\begin{aligned}[\ & paint(car(electric, !suv)) \twoheadrightarrow blue,\\ & paint(car(!diesel, !suv))) \twoheadrightarrow white,\\ & paint(x) \twoheadrightarrow red \ ]\end{aligned}$$

and the formalism we have proposed can be used to transform this latter system into the initial one and eventually into the unordered one.

We consider the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ which is the smallest set containing the set $\mathcal{X}$ of *variable* symbols and such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ whenever $f \in \mathcal{F}$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ for $i \in [1, \ldots, n]$; the set of symbols $\mathcal{F}$ consists of a set $\mathcal{D}$ of *defined symbols* and of a set $\mathcal{C}$ of *constructors*. The linear terms over $\mathcal{T}(\mathcal{C}, \mathcal{X})$ are called *constructor patterns* and the patterns in $\mathcal{T}(\mathcal{C})$ are called *values*.

Given a value $v$ and a constructor pattern $p$, $v$ is an instance of $p$ if there exists a substitution $\sigma$ such that $v = \sigma(p)$ and in this case we say that $p$ *matches* $v$. The instance relation can be defined inductively:

$$\begin{array}{rcll} x & \lessdot & v & x \in \mathcal{X} \\ c(p_1, \ldots, p_n) & \lessdot & c(v_1, \ldots, v_n) & \text{iff } \wedge_{i=1}^{n} p_i \lessdot v_i, c \in \mathcal{C} \end{array}$$

Given a list of patterns $P = [p_1, \ldots, p_n]$, $P$ matches a value $v$ with pattern $p_i$, denoted $P \lessdot_i v$, iff the following conditions hold:

$$\begin{aligned} & p_i \lessdot v \\ & p_j \not\lessdot v, \quad \forall j < i \end{aligned}$$

Note that if $P \lessdot_i v$ then for all $j \neq i$, $P \not\lessdot_j v$.

The *ground semantics* of a constructor pattern $p \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ is the set of all its ground constructor instances: $[\![p]\!] = \{\sigma(p) \mid \sigma(p) \in \mathcal{T}(\mathcal{C})\}$.

Based on these definitions, several problems can be expressed [19], [20]:

- a list of patterns $P$ is *exhaustive* iff for all values $v$ there exists an $i$ such that $P \lessdot_i v$,
- a pattern $p_i \in P$ is *useless* iff there does not exist a value $v$ such that $P \lessdot_i v$,
- the *disambiguation* of a list of patterns $P = [p_1, \ldots, p_n]$ consists in finding sets of patterns $P_1, \ldots, P_n$ such that for each $i \in [1, \ldots, n]$, $[\![P_i]\!] = [\![p_i]\!] \setminus \cup_{j=1}^{i-1} [\![p_j]\!]$.

In order to solve these problems we defined *extended patterns*

$$p \quad := \quad \mathcal{X} \mid c(p_1, \ldots, p_n) \mid p_1 + p_2 \mid p_1 \setminus p_2 \mid \bot$$

Intuitively, $p_1 + p_2$ matches a term if it is matched by one $p_1$ or $p_2$, $p_1 \setminus p_2$ matches a term if it is matched by $p_1$ but not by $p_2$, $\bot$ matches no term.

We also proposed a rewriting system $\mathfrak{R}_\setminus$ transforming any extended pattern into $\bot$ or into a set of constructor patterns [14], and showed that this system is convergent and that it preserves the ground semantics.

To check if a list $[p_1, \ldots, p_n]$ of constructor patterns is exhaustive it is then enough to check that the result of reducing $x \setminus (p_1 + \cdots + p_n)$ using $\mathfrak{R}_\setminus$ leads to $\bot$, meaning that there is no value which can't be matched by one of $p_1, \ldots, p_n$. To check if a pattern $p_i$ in the list is useless it is enough to check that the result of reducing $p_i \setminus (p_1 + \cdots + p_{i-1})$ using $\mathfrak{R}_\setminus$ leads to $\bot$, meaning that there is no value matched by $p_i$ which can't be matched by one of $p_1, \ldots, p_{i-1}$.

As far as it concerns the disambiguation problem for a set $P = [p_1, \ldots, p_n]$, each set $P_i$, for $i \in [1, \ldots, n]$, is obtained by computing using $\mathfrak{R}_\setminus$ the complement $p_i \setminus (p_1 + \cdots + p_{i-1})$. As a consequence, the definition of a function by a list of equations can be replaced by an equivalent one consisting of a set of equations, each equation in the list being replaced by a set of equations corresponding to the set of patterns obtained by disambiguation of the respective equation. The formalism handles also anti-patterns and proceeds carefully with the right-hand side of equations; the interested reader can refer to [14] for the technical details. The list of rules presented at the beginning of the section are transformed into the corresponding set of rules using the formalism.

## III. Static analysis

Now, that an ordered list of rules can be transformed into an equivalent set of rules we can use classical TRS tools in order to verify properties such as the confluence or the termination of the corresponding system. There are also multiple approaches, ranging from model checking ones [21], [22] to tree automata completion [23], that can then be used to verify that the terms obtained by reduction with respect to a given CTRS have (or not) a certain shape.

In the context of program transformation we generally want to guarantee that specific constructor symbols, or patterns, are

absent from the programs obtained by transformation. For this, an approach based on the extended patterns presented in the previous section and relying on a system of annotations of the function symbols by the patterns to be eliminated has been introduced in [15].

For instance, let us consider lists of expressions build out of (wrapped) integers and lists and a transformation which flattens list expressions:

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

To statically ensure that the result of the transformation contains no nested lists, we annotate the function symbol *flatten* with the corresponding (anti-)pattern $p := \mathit{cons}(\mathit{lst}(l_1), l_2)$ and check that the rewriting system is consistent with the annotation [15]. The method relies on an over-approximation of the potential results obtained by reduction and thus, it may lead to some false negatives. For example, the system obtained by replacing the third rule with

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

produces only flat lists but could not be verified with this method.

A more elaborate annotation system which allows for a substantially more precise description of the expected behaviour of a transformation has been proposed in [16]. Here, each function is assigned not only a single (anti-)pattern specifying the post-conditions on the expected outcome but also the pre-conditions guaranteeing the form of the results. The new approach relies on an inference method to characterize the substitutions consistent with the pre-conditions and a verification method guaranteeing that the application of these substitutions is consistent with the post-condition.

For this, we extended our pattern formalism with pattern products: the product $p_1 \times p_2$ matches any term matched by both its components. The instance relation and the ground semantics are extended as expected.

Then, every defined symbol $f$ is associated with a profile $p_1 * \cdots * p_n \mapsto p$ indicating that the normal form of a ground term of the form $f(t_1, \ldots, t_n)$, when it exists, contains no value matching $p$ if the terms $t_1, \ldots, t_n$ can be reduced only to values that don't contain subterms matching the patterns $p_1$-, $\ldots, p_n$ respectively. For simplicity, we consider here only one profile for each symbol although the general formalism allows a set of profiles.

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

The notion of pattern-freeness is then introduced. A value is $p$-free if and only if $p$ matches no subterm of the value. A constructor pattern is pattern-free if all its ground instances are $p$-free. A general term is $p$-free if and only if for all subterms $u$ headed by a defined symbol $f$ of profile $p_1 * \cdots * p_n \mapsto q$, replacing all instances of $u$ in $t$ by any $q$-free value, results in a $p$-free term. Intuitively, this corresponds to considering an over-approximation of the set of potential normal forms of that subterm, and therefore of the whole term. The ground semantics presented in the previous section is extended for any term and is consistent with the pattern-freeness (annotations) of the respective term.

We say that a rewrite rule $l \twoheadrightarrow r$ is *semantics preserving* iff for all substitution $\sigma$, we have $[\![\sigma(r)]\!] \subseteq [\![\sigma(l)]\!]$. A CTRS is semantics preserving iff all its rewrite rules are.

We can than prove that given a semantics preserving CTRS $\mathcal{R}$, if a ground term $t$ reduces with respect to $\mathcal{R}$ into the term $v$, then $[\![v]\!] \subseteq [\![t]\!]$.

Since ground semantics is consistent with pattern-freeness, the semantics preservation allows the characterization of the potential normal forms for a semantics preserving CTRS. We briefly present below sufficient conditions guaranteeing this latter property and a method for automatically checking these conditions.

Given a constructor rewrite rule $f^{\pi}(l_1, \ldots, l_n) \twoheadrightarrow r$ with a profile $\pi = p_1 * \cdots * p_n \mapsto p$, we say that the rule *satisfies the profile* $\pi$ iff for all substitution $\sigma$, $\sigma(l_i)$ is $p_i$-free for all $i \in [1, n] \Longrightarrow \sigma(r)$ is $p$-free.

Profile satisfaction is a necessary and sufficient condition for semantics preservation: a constructor rewrite rule $f^{\pi}(l_1, \ldots, l_n) \twoheadrightarrow r$ is semantics preserving iff it satisfies $\pi$.

To verify that a rewrite rule is semantics preserving we first infer the shapes of the terms that could be used to instantiate the variables in the left-hand side such that the profile of the head symbol is verified and then, check that when replacing accordingly the variables in the right-hand side we respect the post-condition of the profile.

The inference as well as the checking are performed using an approach very similar to the one presented in the previous section. To characterize substitutions $\sigma$ such that $\sigma(l_i)$ is $p_i$-free, we note that $\sigma(l_i)$ is $p_i$-free iff $[\![\sigma(l_i)]\!] \subseteq [\![l_i]\!] \cap [\![x^{-p_i}]\!] = [\![l_i \times x^{-p_i}]\!]$ (with $x^{-p_i}$ a fresh $p_i$-free variable, i.e. whose instances are $p_i$-free). We introduced in [15] an extended version of the rewrite system $\mathfrak{R}_{\setminus}$ to handle products of patterns, and $p$-free variables (using an intermediary algorithm to observe the shape of all subterms of their instances). This extended system is used to reduce the product $l_i \times x^{-p_i}$ into a sum of aliased patterns, *i.e.* patterns where all variables are aliased with a pattern providing a description of the instances verifying the considered pattern-free pre-conditions. The variables are then replaced accordingly in the right-hand side of the rule and a similar approach is used to reduce the obtained term, in order to check the pattern-free post-condition.

The analysis method assumes the linearity of the rules of the CTRS considered. In practice, the aliasing approach to the inference step of the analysis can handle the non-linearity of the left-hand side of the rules quite effectively. However, the over-approximation induced by the notion of pattern-freeness, does not describe accurately non-linear terms in the right-hand

side of these rules. As a first approximation, the static analysis can be applied by linearization of these terms. The method presented in [15] can also use a strictness assumption on the rewriting strategy to provide a more precise analysis, by taking into account some instanciation constraints on the variables of the right-hand side of the rules.

Finally, the current version of the static analysis relies on a more involved notion of pattern-freeness, defined by partially instanciating variables under defined symbols, in order to take into account as many instanciation constraints as possible. Thanks to this updated formalism, the method can provide a much more accurate static analysis of these non-linear cases, while only relying on a confluence assumption.

## IV. Conclusion

We have briefly presented here the formalisms introduced in [14]–[16]. They allow one to compile anti-patterns and ordered lists of rules into classical TRS, and to statically verify properties of the rewrite system and of the terms obtained when reducing using the system. The approaches have been implemented and are publicly available at `http://github.com/plermusiaux/pfree_check`.

We are focusing now on extending the method to handle higher-order functions. The extension is quite straightforward for higher-order functions with a unique annotated profile but becomes more complicated when function symbols could have multiple profiles.

## References


[1] J. Meseguer and C. Braga, "Modular rewriting semantics of programming languages," in *AMAST 2004*, ser. LNCS, 2004, vol. 3116, pp. 364–378.

[2] G. Rosu and T. Serbanuta, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[3] D. Lacey and O. de Moor, "Imperative program transformation by rewriting," in *International Conference on Compiler Construction, CC 2001*, ser. Lecture Notes in Computer Science, R. Wilhelm, Ed., vol. 2027. Springer, 2001, pp. 52–68.

[4] F. Bellegarde, "Program transformation and rewriting," in *International Conference on Rewriting Techniques and Applications, RTA-91*, ser. Lecture Notes in Computer Science, vol. 488. Springer, 1991, pp. 226–239.

[5] J.-P. Jouannaud and C. Kirchner, "Solving equations in abstract algebras: a rule-based survey of unification," in *Computational Logic. Essays in honor of Alan Robinson*. The MIT press, 1991, ch. 8, pp. 257–321.

[6] M. Marin, "Extending Mathematica with Anti-Patterns," in *12th International Mathematice Symposium (IMS 2015)*, 2015, pp. 1–6.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, "The Maude 2.0 System," in *RTA 2003*, ser. LNCS, vol. 2706. Springer-Verlag, 2003, pp. 76–87.

[8] E. Visser, "Strategic pattern matching," in *RTA 1999*, ser. LNCS, vol. 1631. Springer, 1999, pp. 30–44.

[9] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles, "Tom: Piggybacking rewriting on java," in *RTA 2007*, ser. LNCS, vol. 4533. Springer-Verlag, 2007, pp. 36–47.

[10] A. Krauss, C. Sternagel, R. Thiemann, C. Fuhs, and J. Giesl, "Termination of Isabelle functions via termination of rewriting," in *ITP 2011*, ser. LNCS. Springer Berlin Heidelberg, 2011, pp. 152–167.

[11] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann, "Automated termination proofs for Haskell by term rewriting," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 2, pp. 7:1–7:39, 2011.

[12] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann, "Analyzing program termination and complexity automatically with aprove," *Journal of Automated Reasoning*, vol. 58, no. 1, pp. 3–31, 2017.

[13] M. Avanzini, U. Dal Lago, and G. Moser, "Analysing the complexity of functional programs: Higher-order meets first-order," in *ICFP 2015*. ACM, 2015, pp. 152–164.

[14] H. Cirstea and P.-E. Moreau, "Generic encodings of constructor rewriting systems," in *International Symposium on Principles and Practice of Programming Languages, PPDP 2019*. ACM, 2019, pp. 8:1–8:12.

[15] H. Cirstea, P. Lermusiaux, and P.-E. Moreau, "Pattern eliminating transformations," in *International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2020*, ser. Lecture Notes in Computer Science, vol. 12561. Springer, 2020, pp. 74–92.

[16] ——, "Static analysis of pattern-free properties," in *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*. ACM, 2021, pp. 9:1–9:13.

[17] C. Kirchner, R. Kopetz, and P.-E. Moreau, "Anti-pattern matching," in *ESOP 2007*, ser. LNCS, vol. 4421. Springer, 2007, pp. 110–124.

[18] H. Cirstea, C. Kirchner, R. Kopetz, and P.-E. Moreau, "Anti-patterns for rule-based languages," *Journal of Symbolic Computation*, vol. 45, no. 5, pp. 523 – 550, 2010.

[19] L. Maranget, "Warnings for pattern matching," *Journal of Functional Programming*, vol. 17, no. 3, pp. 387–421, 2007.

[20] A. Krauss, "Pattern minimization problems over recursive data types," in *ICFP 2008*. ACM, 2008, pp. 267–274.

[21] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar, "Abstract regular tree model checking," in *International Workshop on Verification of Infinite-State Systems, INFINITY 2005*, ser. Electronic Notes in Theoretical Computer Science, J. Srba and S. A. Smolka, Eds., vol. 149, no. 1. Elsevier, 2005, pp. 37–48.

[22] N. Kobayashi, N. Tabuchi, and H. Unno, "Higher-order multi-parameter tree transducers and recursion schemes for program verification," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*. ACM, 2010, pp. 495–508.

[23] T. Genet, "Towards static analysis of functional programs using tree automata completion," in *International Workshop on Rewriting Logic and Its Applications, WRLA 2014*, ser. Lecture Notes in Computer Science, vol. 8663. Springer, 2014, pp. 147–161.