WADT'04, LNCS 3423, pp 218-233. 2004

From Conditional to Unconditional Rewriting

Grigore Roşu

Department of Computer Science,
University of Illinois at Urbana-Champaign.
grosu@cs.uiuc.edu

Abstract. An automated technique to translate conditional rewrite rules into unconditional ones is presented, which is suitable to implement, or compile, conditional rewriting on top of much simpler and easier to optimize unconditional rewrite systems. An experiment performed on world's fastest conditional rewriting engines shows that speedups for conditional rewriting of an order of magnitude can already be obtained by applying the presented technique as a front-end transformation.

1 Introduction

Conditional term rewriting is a crucial paradigm in the algebraic specification of abstract data types, since it provides a natural means for executing equational specifications. Many specification languages today, including Maude [4], ELAN [3], OBJ [9], CafeOBJ [6], provide conditional rewriting engines to allow users to execute and reason about specifications. Conditional rewriting also plays a foundational role in functional logic programming [10]. Additionally, there are many researchers, including the author, considering rewriting a powerful programming paradigm by itself, who are often frustrated that conditional rewrite "programs" are significantly slower than unconditional ones doing the same thing.

Conditional rewriting is, however, rather inconvenient to implement directly. To reduce a term, a rewriting engine needs to maintain a *control context* for each conditional rule that is tried. Due to the potential nesting of rewrite rule applications, such a control context may grow arbitrarily. Our technique automatically translates conditional rewrite rules into unconditional rules, by *encoding the necessary control context into data context*. The obtained rules can be then executed on (almost) any unconditional rewriting engine, whose single task is to *match-and-apply* unconditional rules. Such a simplified engine can be seen as a *rewrite virtual machine*, which can be even implemented in hardware for increased efficiency, and our transformation technique can be seen as a compiler.

Experiments performed on two fast rewriting engines show that speedups of an order of magnitude can be obtained right now if one uses our transformation technique as a front-end. However, since these rewrite engines are optimized for conditional rewriting, we expect significant further increases in efficiency if one just focus on the much simpler problem of developing optimized unconditional rewrite engines and use our technique as a front-end. Even though presented as a translation of conditional rewrite systems into unconditional ones, our technique

can easily be adapted and used as a means to implement conditional rewriting also without applying an explicit transformation. We will discuss this elsewhere.

The proofs of Proposition 1 and Theorem 1 can be found in [16].

Related work. Stimulated by the benefits of transforming conditional term rewrite systems (CTRSs) into equivalent unconditional term rewrite systems (TRSs), there has been much research on this topic. Despite the apparent simplicity of most transformations, they typically work for restricted CTRSs and their correctness, when they are correct, is quite technical and tricky to prove. A large body of literature has been dedicated to transformations preserving only certain properties of CTRSs, e.g., termination and/or confluence. We do not discuss these here; the interested reader is referred, e.g., to Ohlebusch [14].

In this paper we focus on transformations that generate TRSs *computationally equivalent* to CTRSs, i.e., the TRSs can be *transparently* used to reduce terms in the original CTRSs. The first attempt in this category is due to Bergstra and Klop [2], for a restricted class of CTRSs (whose underlying unconditional TRS is left-linear and without superposition); unfortunately, this transformation was shown to be unsound by Dershowitz and Okada [5]. The transformation in Giovannetti and Moiso [8] works only under severe restrictions on the original CTRS: no superposition, simply terminating (enforced by the requirement of a simplification ordering), and non-overlapping of conditions with left-hand-side (lhs) terms. Hintermeier [11] proposes a technique where an "interpreter" for CTRS is defined as a TRS, providing explicit rewrite definitions for matching and applications of rewrite rules. Besides being technically very intricate and practically inefficient, this transformation is proven to be correct only when the original CTRS is confluent and strictly terminating (i.e., decreasing). Our work in this paper was motivated by efforts in *rewriting logic semantics* [12], where rewriting logic is used as a core mechanism to give operational semantics to concurrent programming languages. In this framework, as well as in many others, restrictions such as termination and/or confluence are unacceptably strong. Indeed, in any programming language there are programs which do not terminate, and concurrency leads quickly to non-confluence (e.g., data-races).

Our technique was presented at WADT'04 and was developed independently from that of Viry [17]. However, the two techniques have many similarities[1]. They are both based on decorations of terms, obtained by adding as many auxiliary arguments to each operation f as conditional rules in the original CTRS having f at the top of their lhs. The procedure in [17] encodes the condition of each rule within a special data-structure that occurs as the corresponding auxiliary argument associated to the operation occurring at the top of its lhs. Two unconditional rules are added in the generated TRS for each conditional rule in the original CTRS, one for initializing the special data-structure and the other for continuing the rewriting process when the condition was evaluated. For example, the CTRS (taken from [17]) $\mathcal{R}$ below is transformed into $\mathcal{R}'$:

[1] We thank Bernhard Gramlich for making us aware of Viry [17].

$$(\mathcal{R}) \begin{cases} f(g(x)) \to p(x) \texttt{ if } c(x) \to^* true \\ f(h(x)) \to q(x) \texttt{ if } d(x) \to^* true \\ c(a) \to true \end{cases} \quad (\mathcal{R}') \begin{cases} f(g(x) \mid \bot, z) \to f(g(x) \mid [c(x), (x)], z) \\ f(x \mid [true, (y)], z) \to p(y) \\ f(h(x) \mid z, \bot) \to f(h(x) \mid z, [d(x), (x)]) \\ f(x \mid z, [true, (y)]) \to q(y) \\ c(a) \to true \end{cases}$$

where "|" is syntactic sugar for ",", separating the normal arguments from the auxiliary ones; "$\bot$" is a special constant whose occurrence states that the corresponding conditional rule has not been tried yet on the current position; a structure $[u, \vec{s}]$ occurring during a rewriting sequence as an auxiliary argument of an operation, means that u is the current reduction status of the corresponding condition that started to be evaluated at some point, and that $\vec{s}$ was the substitution at that point that allowed the lhs of that rule to match. The substitution is needed by the second unconditional rule associated to a conditional rule, to correctly initiate the reduction of the rhs of the original conditional rule.

Despite being proved sound and complete by Viry [17], the procedure above, unfortunately, cannot be used *as is* to interpret any CTRS on top of a TRS. That is because it destroys the confluence of the original CTRS, thus leading to normal forms in the TRS which can be further reduced in the CTRS. Indeed, let us consider the following CTRS $\mathcal{R}$, from Antoy, Brassel and Hanus [1], together with Viry's transformation $\mathcal{R}'$:

$$(\mathcal{R}) \begin{cases} f(g(x)) \to x \texttt{ if } x \to^* 0 \\ g(g(x)) \to g(x) \end{cases} \quad (\mathcal{R}') \begin{cases} f(g(x) \mid \bot) \to f(g(x) \mid [x, (x)]) \\ f(x \mid [0, (y)]) \to y \\ g(g(x)) \to g(x) \end{cases}$$

$\mathcal{R}$ is confluent but $\mathcal{R}'$ is not: $f(g(g(0)) \mid \bot)$ can be reduced to both 0 and $f(g(0) \mid [g(0), (g(0))])$; the latter occurs because the "conditional" rule is first tried and "failed", then the "unconditional" one is applied successfully thus changing the context so that the "conditional" rule becomes conceptually applicable, but it fails to apply since it was already marked as "tried". To solve this problem, Viry [17] proposes a reduction strategy within the generated TRS, called *conditional eagerness*, stating that $t_1, ..., t_n$ must be already in normal form before a "conditional" rule can be applied on a term $f(t_1, ..., t_n \mid \bot, ..., \bot)$. This way, in the example above, $g(g(0))$ is enforced to be first evaluated to $g(0)$ and only then $f(g(0) \mid \bot)$ is applied the "conditional" rule and eventually reduced to 0. However, conditional eagerness does not seem to be trivial to enforce in an unconditional rewriting engine, unless that is internally modified. One simple, but very restrictive, way to ensure conditional eagerness is to enforce innermost rewriting both in the original CTRS and in the resulting TRS.

A different fix to Viry's technique was proposed by Antoy, Brassel and Hanus [1], namely to restrict the input CTRSs to *constructor-based* ones, i.e., ones in which the operations are split into *constructors* and *defined*, and the lhs of each rule is a term of the form $f(t_1, ..., t_n)$, where f is defined and $t_1, ..., t_n$ are all

constructor terms. The problematic CTRS above is *not* constructor-based, so Viry's procedure is not guaranteed to work correctly on it. While constructor-baseness is an easy to check and automatic correctness criterion, we believe that it is an unnecessary strong restriction on the input CTRS, which may make the translation useless in many situations of practical interest.

An additional drawback of Viry's transformation is that it increases the number of rewrite rules having the same operator at the root of their lhs, which tends to be a source of matching overhead on many rewrite engines, especially in the context of very large CTRSs[2]. Therefore, we are still left with no satisfactory translation of CTRSs into equivalent TRSs. In this paper we give a practical solution to this problem, which imposes *no restrictions* on the original CTRS, which adds exactly one unconditional rule for each conditional rule in the original CTRS, and which is shown to bring a significant speedup on current conditional rewrite engines if applied as a front-end transformation. Our translation is *almost* ideal, in that it still requires some special support from the underling unconditional rewrite engine: to provide (1) a binary equality operation, denoted $equal?(t, t')$ in this paper, returning *true* iff the normal forms of t and t' are identical, and (2) a conditional $if(b, t, t')$ which is eager in b and lazy in t and t'. However, all rewriting engines that we know provide them [9, 3, 4, 6, 18]. They can also be easily defined if the rewriting engine provides support for simple contextual strategies, which all rewriting engines that we know do.

2 Preliminaries

We recall some basic notions of conditional rewriting, referring the interested reader to [14] for more details. An (unsorted) *signature* Σ is a finite set of operational symbols, each having zero or more arguments. We let $\Sigma_n \subseteq \Sigma$ denote the set of operations of n arguments. The operations of zero arguments in Σ_0 are called *constants*. We assume an infinite set of *variables* $\mathcal{X}$. Given a signature Σ and a set of variables $X \subseteq \mathcal{X}$, we let $T_\Sigma(X)$ denote the algebra of *Σ-terms* over variables in X. A term without variables is called *ground*. A map $\theta : \mathcal{X} \to T_\Sigma(\mathcal{X})$ can be uniquely extended to a morphism of algebras $T_\Sigma(\mathcal{X}) \to T_\Sigma(\mathcal{X})$ replacing each variable in x by a term $\theta(x)$; to keep the notation simple, we let θ also denote this map. A conditional Σ-rewrite rule has the form

$$l \to r \texttt{ if } u_1 = v_1, \cdots, u_m = v_m,$$

where l, r, u_1, v_1, ..., u_m, v_m are Σ-terms in $T_\Sigma(\mathcal{X})$. The term l is called the *left-hand-side (lhs)*, r is called the *right-hand-side (rhs)*, and $u_1 = v_1, \cdots, u_m = v_m$ is called the *condition* of the rewriting rule above. As usual, we disallow rewriting rules whose lhs is a variable. Further, we assume that the lhs of a rewriting rule contains all the variables that occur in that rule, that is, following the terminology in [13] our rewrite systems are *of type 1*. If $m = 0$, the rewrite

[2] We have encountered CTRSs of thousands of rules in the context of rewriting logic semantics of programming languages [12]

rule is called *unconditional* and written $l \to r$. Unless specified differently, by conditional rule we mean a rule with $m \geq 1$. A *conditional (unconditional) Σ-term rewrite system* $\mathcal{R} = (\Sigma, R)$, abbreviated *CTRS (TRS)*, consists of a finite set R of conditional (unconditional) Σ-rewrite rules. Any Σ-rewrite system $\mathcal{R} = (\Sigma, R)$ generates a relation $\to_{\mathcal{R}}$ on $T_\Sigma(\mathcal{X})$, defined recursively as follows. For any $\theta : \mathcal{X} \to T_\Sigma(\mathcal{X})$, $t[\theta(l)] \to_{\mathcal{R}} t[\theta(r)]$ whenever there exists some s_i such that $\theta(u_i) \to^\star_{\mathcal{R}} s_i$ and $\theta(v_i) \to^\star_{\mathcal{R}} s_i$ for any $1 \leq i \leq m$, where t is a term having one occurrence of a special variable, say $*$, $t[\theta(l)]$ is the term obtained by substituting $*$ with $\theta(l)$ in t, and $\to^\star_{\mathcal{R}}$ is the reflexive and transitive closure of $\to_{\mathcal{R}}$. Hence, $\alpha \to_{\mathcal{R}} \beta$ iff α has a subterm matching the lhs of a rule in R via some substitution, s.t. all the terms in each equality in the condition can be iteratively reduced to a common term. Such CTRSs are also called *join* or *standard* [14]. Alternative interpretations of equalities are also possible, and we will discuss transformations of those elsewhere soon. However, as their name suggests, standard conditional rewrite systems are the most common ones and major rewriting engines, e.g., Maude [4] and ELAN [3], support them. These systems perform millions of rewrites per second on standard PCs and are, at our knowledge, the fastest rewriting engines.

Terms which cannot be reduced any further in $\mathcal{R}$ are called *normal forms* for $\mathcal{R}$. Rewriting of a given term may not terminate for two reasons: either the reduction of the condition of a rule does not terminate, or there are some rules that can be applied infinitely often on the given term. On systems like Maude or ELAN, the effect in both situations is the same: the system loops forever unless it crashes running out of memory. Because of this reason, we do not make any distinction between the two causes, and simply call a Σ-rewriting system *terminating* iff it always reduces any Σ-term to a normal form (we let this notion at an intuitive level here, but it can be formalized). Letting $\_;\_$ denote the composition of relations, a relation $\to$ is *confluent* iff $\leftarrow^\star; \to^\star \subseteq \to^\star; \leftarrow^\star$.

3 Defining the Basic Infrastructure

We define several operators together with appropriate (unconditional) rules. Most rewriting engines have these basic operators built-in, but here we do not assume any existing operators and therefore define everything needed.

Let *true* and *false* be two constants which are assumed not defined within any given CTRS (otherwise change their name). Let us also assume a fresh binary operator $\wedge$, written in infix associative notation, together with the rules:

$$\begin{array}{ll} \mathit{true} \wedge \mathit{true} \ \to \mathit{true}, & \mathit{true} \wedge \mathit{false} \ \to \mathit{false}, \\ \mathit{false} \wedge \mathit{true} \ \to \mathit{false}, & \mathit{false} \wedge \mathit{false} \ \to \mathit{false}. \end{array}$$

These will be needed to evaluate conditions that will be translated into corresponding conjunctions of equalities; equalities will be defined shortly.

Let us now consider a special operator $\mathit{if}(\_,\_,\_)$, together with the rules:

$$\mathit{if}(\mathit{true}, x, y) \ \to \ x, \qquad \mathit{if}(\mathit{false}, x, y) \ \to \ y.$$

This operator is assumed *eager in its first argument and lazy in the others*. Most rewrite engines provide it as builtin, so the two rules above are not needed.

We need another special operator, $equal?(\_,\_)$, that reduces its arguments and returns *true* if they are identical and *false* otherwise. One obvious rule to add is $equal?(x,x) \to true$. Moreover, for all $\sigma \in \Sigma_n$ we add

$$equal?(\sigma(x_1,\ldots,x_n),\sigma(y_1,\ldots,y_n)) \to equal?(x_1,y_1)\wedge\cdots\wedge equal?(x_n,y_n), \quad (1)$$

where $x_1,...,x_n,y_1,...,y_n$ are disjoint variables. These rules propagate the equality of two terms having the same operator as root to the equality of their corresponding sub-terms. Note that σ may be a constant in Σ_0, in which case, by convention, $equal?(x_1,y_1)\wedge\cdots\wedge equal?(x_n,y_n)$ is *true*, the unit of $\wedge$. The following rules, one for each pair $\sigma \in \Sigma_n$, $\tau \in \Sigma_m$ of different operations in Σ, state that terms having different operations at root are not equal:

$$equal?(\sigma(x_1,\ldots,x_n),\tau(y_1,\ldots,y_m)) \to false.$$

Note that *equal?* needs to be *eager in both its arguments*. All rewrite engines we know have such an operator builtin, so these rules are not needed in practice.

For a given signature Σ, let Σ' denote the signature Σ extended with all the auxiliary operations above, and let $\mathcal{I}(\Sigma)$ be the Σ'-rewriting system containing all the rules above. We call $\mathcal{I}(\Sigma)$ the *infrastructure rewriting system of* Σ.

Proposition 1. *Let $\mathcal{R}$ be a Σ-rewrite system, conditional or not. Then*

1. *$\mathcal{I}(\Sigma)$ is a confluent and terminating unconditional Σ'-rewrite system;*
2. *If $u,v \in T_\Sigma(X)$ then $u\ (\to_{\mathcal{R}}^{\star};\leftarrow_{\mathcal{R}}^{\star})\ v$ iff $equal?(u,v) \to_{\mathcal{R}\cup\mathcal{I}(\Sigma)}^{\star} true$;*
3. *If u,v are ground Σ-terms then a normal form of $equal?(u,v)$ in $\mathcal{R}\cup\mathcal{I}(\Sigma)$ is either true or false;*
4. *$\mathcal{R}$ terminates if and only if $\mathcal{R}\cup\mathcal{I}(\Sigma)$ terminates;*
5. *If $\mathcal{R}$ is confluent and terminates, i.e., it has unique normal forms, then $\mathcal{R}\cup\mathcal{I}(\Sigma)$ is also confluent and terminates.*

By *2.*, one can replace any equality $u = v$ in the condition of a rule in $\mathcal{R}$ by $equal?(u,v) = true$. Note that the restriction on u and v to be ground is crucial in *3.* Suppose, e.g., that u is a variable, say x. Then there is no rule to reduce the term $equal?(x,v)$ to *true* or *false*. Moreover, one does *not* want to add rules of the form $equal?(x,\tau(y_1,\ldots,y_m)) \to false$ to $\mathcal{I}(\Sigma)$ because one would destroy the confluence of $\mathcal{I}(\Sigma)$ and thus the correctness of the definition of *equal?*: indeed, $equal?(\tau(y_1,\ldots,y_m),\tau(y_1,\ldots,y_m))$ would reduce to both *true* and *false* in $\mathcal{I}(\Sigma)$.

4 The Main Transformation

The major reason for which conditional rules are inconvenient to implement in a rewriting engine is because, in order to reduce a term, the rewriting engine needs to maintain a *control context* for each conditional rule that is tried to be

applied. By control context we here mean the status of the evaluation of the condition (note that a condition is a set of equalities) plus the right hand term that needs to replace the left hand one in case the condition evaluates to *true*. Due to the potential nesting of rewrite rule applications, such a control context may grow arbitrarily, meaning that the rewriting engine needs to pay special care to choosing appropriate data-structures to maintain it and to recover the computation in case the evaluation of a condition fails.

Example 1. Let us consider natural numbers built with 0 and successor s, together with the following, on purpose inefficient, conditional rules defining *odd* and *even* operators on natural numbers:

$$\begin{array}{ll} odd(0) \rightarrow 0, & even(0) \rightarrow s(0), \\ odd(s(x)) \rightarrow 0 \ \texttt{if} \ even(x) = 0, & even(s(x)) \rightarrow 0 \ \texttt{if} \ odd(x) = 0, \\ odd(s(x)) \rightarrow s(0) \ \texttt{if} \ even(x) = s(0), & even(s(x)) \rightarrow s(0) \ \texttt{if} \ odd(x) = s(0). \end{array}$$

In order to check whether a natural number n, i.e., a term consisting of n successor operations applied to 0, is odd, a rewriting engine may need $\mathcal{O}(2^n)$ rewrites in the worst case. Indeed, if $n > 0$ then either the second or the third rule of *odd* can be applied at the first step; however, in order to apply any of those rules one needs to reduce the even of the predecessor of n, twice. Iteratively, the evaluation of each even involves the reduction of two odds, and so on. Moreover, the rewriting engine needs to maintain a control context data-structure, storing the status of the application of each (nested) rule that is being tried in a reduction. It is the information stored in this control context that allows the rewriting engine to backtrack and find an appropriate rewriting sequence. □

A challenging question motivating the present work is the following: would it be possible to automatically replace conditional rules like the above by unconditional ones, so that a rewriting engine's single job would be to *match-and-apply* rules, without worrying about any control context aspects? A positive answer to this question could potentially lead to a new generation of efficient rewriting engines, which would take advantage of today's increasingly highly parallel computing architectures and would potentially allow optimizations that were not possible for conditional rewriting. In this section we show how a conditional rewrite system $\mathcal{R}$ can be automatically transformed into an unconditional one $\overline{\mathcal{R}}$, which practically preserves all the properties of $\mathcal{R}$. The major idea is, like in the use of continuations (see [15] for a discussion on several independent discoveries of continuations, and [7] for a pragmatic presentation of continuation), to convert the control context into data context. This way, the term to be rewritten is enriched at appropriate positions to contain *all* the information needed to continue its reduction. The rewriting engine does not need to maintain any auxiliary information about the status of the rewriting process: it only needs to find a redex in the term to rewrite and apply a corresponding unconditional rewrite rule, a simple process amenable to high parallelization and optimization.

4.1 An Unsatisfactory Transformation

Once one generates the infrastructure (unconditional) Σ'-rewrite system $\mathcal{I}(\Sigma)$, a simple-minded way to transform a conditional Σ-rewrite system R into an unconditional one is to translate each conditional rule

$$l \ \rightarrow \ r \ \texttt{if} \ \ u_1 = v_1, \cdots, u_m = v_m$$

into an unconditional rewrite rule

$$l \ \rightarrow \ \mathit{if}(\mathit{equal?}(u_1, v_1) \wedge \cdots \wedge \mathit{equal?}(u_m, v_m), r, l).$$

Such a transformation has the desirable property that both the conditional rewrite system and its unconditional variant can "reach", by reduction in zero or more steps starting with a given Σ-term, the same set of Σ-terms. In other words, if a and b are Σ-terms then $a \rightarrow^\star b$ in the conditional Σ-rewrite system if and only if $a \rightarrow^\star b$ in the unconditional Σ'-rewrite system. Therefore, if reachability analysis is what one is interested in then this simple translation provides an effective method to reduce the problem to unconditional rewrite systems. This rewrite system transformation can be useful in systems like Maude, providing commands of the form "`search a =>* b`" searching for a sequence of applications of rewrite rules transforming a into b.

However, this translation cannot be used to execute conditional rewriting on top of an unconditional rewriting engine. Indeed, if the conjunction of equalities reduces to *false* then the unconditional rewrite system leads to an infinite rewriting sequence, by keeping applying the rule above. Would it be possible to properly *mark* the term to rewrite whenever a rule is tried and its condition reduces to *false*, so that that rule will not be applied anymore on that position?.

4.2 Adding Control Context Arguments

Like in Viry [17], the idea is to add a few auxiliary arguments to some operators to keep the necessary control context information. This way, terms to rewrite will store information about the conditional rules that can be potentially applied on each of their subterms. Let $\mathcal{R} = (\Sigma, E)$ be any Σ rewriting system. For each n and each $\sigma \in \Sigma_n$, let us associate a unique number between 1 and k_σ to each conditional rewrite rule in R whose lhs is rooted in σ, that is, a rule of the form

$$\sigma(t_1, \ldots, t_n) \ \rightarrow \ r \ \texttt{if} \ \ u_1 = v_1, \cdots, u_m = v_m,$$

with $t_1, \ldots, t_n, r, u_1, v_1, \ldots, u_m, v_m$ terms and $m \geq 1$, where k_σ is the total number of such rules. Note that k_σ is 0 if there is no rule having σ as a root of its lhs, or if all such rules are unconditional.

Let us next define a signature $\overline{\Sigma}$, replacing each $\sigma \in \Sigma_n$ by an operator of $n + k_\sigma$ arguments, $\overline{\sigma} \in \overline{\Sigma}_{n+k_\sigma}$. The additional k_σ arguments are written at the right of the other n arguments, and they can take only two possible values (or constant terms): *true* or *false*. An important step in our transformation

technique is to replace all the operations in Σ by corresponding operations in $\overline{\Sigma}$. The intuition for the additional arguments comes from the overall idea of passing the control context (due to conditional rules) into data context: the additional i-th argument of an operation $\overline{\sigma}$ staying at some position in a term to rewrite, tells whether the i-th rule having σ at the root of its lhs is enabled or not at that position; if *true* then it means that the rule can potentially be applied, and if *false* then it means that the rule has been already tried at that position but its condition failed to evaluate to *true*, so there is no need to try it anymore. Let us extend this to Σ-terms, by letting the variables unchanged and replacing each operator σ by $\overline{\sigma}$ with the k_σ additional arguments all *true*. Formally, let $\overline{\cdot} : T_\Sigma(\mathcal{X}) \to T_{\overline{\Sigma}}(\mathcal{X})$ be a map from Σ-terms to $\overline{\Sigma}$-terms defined inductively as

- $\overline{x} = x$ for any variable $x \in \mathcal{X}$, and
- $\overline{\sigma(t_1, \ldots, t_n)} = \overline{\sigma}(\overline{t_1}, \ldots, \overline{t_n}, \mathit{true}, \ldots, \mathit{true})$ for any $\sigma \in \Sigma_n$ and any terms $t_1, \ldots, t_n \in T_\Sigma(\mathcal{X})$.

Let's define another useful map from Σ-terms to $\overline{\Sigma}$-terms, $\widetilde{\cdot}^X : T_\Sigma(X) \to T_{\overline{\Sigma}}(\mathcal{X})$, but this time indexed by a finite set of variable $X \subseteq \mathcal{X}$, as follows:

- $\widetilde{x}^X = x$ for any variable $x \in X$, and
- $\widetilde{\sigma(t_1, \ldots, t_n)}^X = \overline{\sigma}(\widetilde{t_1}^X, \ldots, \widetilde{t_n}^X, b_1, \ldots, b_{k_\sigma})$ for any $\sigma \in \Sigma_n$ and any terms $t_1, \ldots, t_n \in T_\Sigma(X)$, where $b_1, \ldots, b_{k_\sigma} \in \mathcal{X} - X$ are some arbitrary but fixed different fresh variables that do not occur neither in X nor in $\widetilde{t_1}^X, \ldots, \widetilde{t_n}^X$.

Therefore, $\widetilde{t}^X$ transforms the Σ-term t into a $\overline{\Sigma}$-term, replacing each operation $\sigma \in \Sigma$ by $\overline{\sigma} \in \overline{\Sigma}$ and adding distinct variables for the additional arguments, following some arbitrary but deterministic conventions. Given a Σ-term t in $T_\Sigma(X)$ of the form $\sigma(t_1, \ldots, t_n)$ for some operation $\sigma \in \Sigma_n$, and given a natural number i between 1 and k_σ, then we let $\widetilde{t}^X_{i/true}$ denote the $\overline{\Sigma}$-term $\overline{\sigma}(\widetilde{t_1}^X, \ldots, \widetilde{t_n}^X, b_1, \ldots, b_{i-1}, \mathit{true}, b_{i+1}, \ldots, b_{k_\sigma})$, that replaces b_i in $\widetilde{t}^X$ by *true*. Similarly, $\widetilde{t}^X_{i/false}$ denotes $\overline{\sigma}(\widetilde{t_1}^X, \ldots, \widetilde{t_n}^X, b_1, \ldots, b_{i-1}, \mathit{false}, b_{i+1}, \ldots, b_{k_\sigma})$, that replaces b_i in $\widetilde{t}^X$ by *false*. Thus, $\widetilde{t}^X_{i/true}$ (resp. $\widetilde{t}^X_{i/false}$) contains the additional control context information whether the i-th conditional rule of σ is enabled.

4.3 An Almost Correct Transformation

For a given conditional Σ-rewrite system $\mathcal{R}$, we can now define an unconditional $\overline{\Sigma}'$-rewrite system[3] $\mathcal{R}'$ by adding to $\mathcal{I}(\overline{\Sigma})$ the following unconditional $\overline{\Sigma}'$-rewrite rules. For each conditional ($m \geq 1$) rule $l \to r$ `if` $u_1 = v_1, \cdots, u_m = v_m$ over variables X in $\mathcal{R}$, say the i-th among the conditional rewrite rules in $\mathcal{R}$ having the root operation of l as a root of their lhs, add to $\mathcal{R}'$ the unconditional rule

$$\widetilde{l}^X_{i/true} \to \mathit{if}\,(\mathit{equal?}(\overline{u_1}, \overline{v_1}) \wedge \cdots \wedge \mathit{equal?}(\overline{u_m}, \overline{v_m}), \overline{r}, \widetilde{l}^X_{i/false}).$$

[3] Note that $\overline{(\Sigma')} = (\overline{\Sigma})'$, so we take the liberty to denote this signature $\overline{\Sigma}'$.

For each unconditional rewrite rule $l \to r$ in $\mathcal{R}$ over variables X, add to $\mathcal{R}'$ an unconditional rewriting rule $\widetilde{l}^X \to \overline{r}$.

Therefore, for each conditional rule in $\mathcal{R}$ we add an unconditional one in $\mathcal{R}'$, whose corresponding additional argument of its transformed lhs is *true*. By throwing the control context's ball into matching's court, this intuitively says that such a rule can be applied on a (sub)term only if it is "enabled" in that (sub)term. Its rhs term has a conditional operation at its root, which first evaluates the conjunction of all the equalities of pairs of terms occurring in the condition of the conditional rule; note that these terms are properly transformed into $\overline{\Sigma}$-terms enabling all possible rules at any of their positions. If the conjunction evaluates to *true* then the rhs of the conditional rule is returned, also modified to enable all possible rules on it. If the condition reduces to *false* then the only thing to do is to "disable" this current rule. Due to the change of the corresponding argument from *true* to $false$, note that matching will disallow this rule to be applied anymore on that (sub)term. Since unconditional rules are always enabled, they are transformed into unconditional rules ignoring the control context arguments in the lhs and enabling all possible rules on its rhs. Note that $\mathcal{R}'$ does not modify any rule in $\mathcal{R}$ if $\mathcal{R}$ already contains only unconditional rules.

Example 2. Let us apply the translation technique above on the conditional rewriting system for odd/even in Example 1. Since there are two conditional rules whose root of lhs is *odd* and two whose root of lhs is *even*, each of these operators will be enriched with two additional arguments. The new, unconditional rewriting system is then:

$$
\begin{array}{l}
\overline{odd}(0, b_1, b_2) \ \to\ 0, \\
\overline{odd}(s(x), true, b_2) \ \to\ if(equal?(\overline{even}(x, true, true), 0),\ 0,\ \overline{odd}(s(x), false, b_2)), \\
\overline{odd}(s(x), b_1, true) \ \to\ if(equal?(\overline{even}(x, true, true), s(0)),\ s(0),\ \overline{odd}(s(x), b_1, false)), \\
\overline{even}(0, b_1, b_2) \ \to\ s(0), \\
\overline{even}(s(x), true, b_2) \ \to\ if(equal?(\overline{odd}(x, true, true), 0),\ 0,\ \overline{even}(s(x), false, b_2)), \\
\overline{even}(s(x), b_1, true) \ \to\ if(equal?(\overline{odd}(x, true, true), s(0)),\ s(0),\ \overline{even}(s(x), b_1, false)).
\end{array}
$$

The unconditional rule for $\overline{odd}$ says that 0 is not an odd number, regardless of the control context. The first conditional rule for $\overline{odd}$ has the constant *true* as the first auxiliary argument of its lhs, telling the matching procedure that this rule can be applied only if it was not previously disabled. If the condition of *if* evaluates to *true* then 0 is returned, otherwise the same term as the lhs, except that *true* is replaced by *false*, thus disabling the current conditional rule to avoid getting into non-terminating rewriting. The variable argument b_2 says that it does not matter whether the second conditional rule is enabled or not (but this information will be preserved in case the first conditional rule is disabled). The other conditional equations are similar. If one wants to test whether a number n, i.e., n consecutive applications of successor on 0, is odd, one should reduce the term $\overline{odd(n)}$, i.e., $odd(n, true, true)$, under the unconditional rewrite system. Note that the operations 0 and s are not added auxiliary arguments because they do not occur as a root of a lhs of any conditional rule in the original conditional rewriting system. □

Unfortunately, the translation above suffers from the same problem as that of Viry [17]: for some CTRSs, the generated TRSs have additional normal forms corresponding to terms which could be further reduced in the original CTRS.

Example 3. Consider the problematic CTRS from Section 1:

$$f(g(x)) \to x \ \texttt{if}\ x = 0,$$
$$g(g(x)) \to g(x),$$

whose corresponding TRS, according to the transformation above, is:

$$f(g(x), true) \to if(equal?(x, 0),\ x,\ f(g(x), false)),$$
$$g(g(x)) \to g(x).$$

Then note that even if $f(g(g(0)))$ admits a unique normal form in the original CTRS, $f(g(g(0)), true)$ admits two normal forms in its corresponding TRS:

$$f(g(g(0)), true) \to f(g(0), true) \to if(equal?(0, 0),\ 0,\ f(g(0), false)) \to^* 0,$$
$$f(g(g(0)), true) \to if(equal?(g(0), 0),\ g(0),\ f(g(g(0)), false)) \to^*$$
$$\to^* f(g(g(0)), false) \to f(g(0), false).$$

The latter cannot be further reduced with the rules in the TRS. □

The problem here, like in Viry's transformation [17], is that a successful application of a rewrite rule may enable some application of a conditional rule that has already been tried before, but at that time failed to apply. One unsatisfactory way to fix this problem is, like in [17], to enforce conditional eagerness on the generated TRS; another, even more unsatisfactory, is to reduce the applicability of the transformation to only innermost, or eager, CTRSs. We next show how to fix this problem in general.

4.4 The Correct Transformation

To fix the problem in the previous subsection, we need a mechanism to "inform" the term to reduce, after each successful application of a rewrite rule, that some "conditional" rules that have been tried before and failed may succeed now. More precisely, we need to traverse the term along the path from the current position (where the successful rule was applied) to its root, and make all the auxiliary arguments of the operations on this path *true*. This can be accomplished, for example, by considering a new (unary) operator, say $\{\_\}$, stating that the enclosed term has just been modified, together with appropriate rewrite rules to propagate this information upwards, updating the "applicability bits": for each $\sigma \in \Sigma_n$ and each $1 \leq i \leq n$, consider a rule

$$\sigma(x_1, ..., x_{i-1}, \{x_i\}, x_{i+1}, ..., x_n, b_1, ..., b_{k_\sigma}) \to$$
$$\to \{\sigma(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n, true, ..., true)\}.$$

The applicability information of an operation can be updated from several of its subterms; to keep this operation idempotent, we also consider the rule

$$\{\{x\}\} \to \{x\}.$$

Formally, for a given conditional Σ-rewrite system $\mathcal{R}$, we let $\overline{\Sigma}'_{\{\}}$ define the signature $\overline{\Sigma}'$ in the previous subsection extended with the unary operator $\{\_\}$ above, and we let $\overline{\mathcal{R}}$ be the unconditional $\overline{\Sigma}'_{\{\}}$-rewrite system extending $\mathcal{I}(\overline{\Sigma})$ with the operator $\{\_\}$ together with its unconditional rewrite rules above, as well as with the following rules. For each conditional ($m \geq 1$) rule $l \to r$ `if` $u_1 = v_1, \cdots, u_m = v_m$ over variables X in $\mathcal{R}$, say the i-th among the conditional rewrite rules in $\mathcal{R}$ having the root operation of l as a root of their lhs, add to $\overline{\mathcal{R}}$:

$$\widetilde{l}^X_{i/true} \to \mathit{if}(\mathit{equal?}(\{\overline{u_1}\},\{\overline{v_1}\}) \wedge \cdots \wedge \mathit{equal?}(\{\overline{u_m}\},\{\overline{v_m}\}), \{\overline{r}\}, \widetilde{l}^X_{i/false}).$$

For each unconditional rewrite rule $l \to r$ in $\mathcal{R}$ over variables X, add to $\overline{\mathcal{R}}$ an unconditional rewriting rule $\widetilde{l}^X \to \{\overline{r}\}$.

Before we formalize the exact relationship between CTRSs and their unconditional variants, let us define another map of terms, this time from $\overline{\Sigma}$-terms to Σ-terms. Let $\widehat{\cdot}\colon T_{\overline{\Sigma}}(\mathcal{X}) \to T_{\Sigma}(\mathcal{X})$ be the map defined inductively as

- $\widehat{x} = x$ for any variable $x \in \mathcal{X}$, and
- $\overline{\sigma}(t'_1, \ldots, \widehat{t'_n, s_1}, \ldots, s_{k_\sigma}) = \sigma(\widehat{t'_1}, \ldots \widehat{t'_n})$ for any operator $\sigma \in \Sigma_n$ and any terms $t'_1, \ldots, t'_n, s_1, \ldots, s_{k_\sigma} \in T_{\overline{\Sigma}'}(\mathcal{X})$.

Therefore, $\widehat{t'}$ forgets all the auxiliary arguments of each operation occurring in t'. Note in particular that $\widehat{\overline{t}} = \widehat{\widetilde{t}} = t$ for any $t \in T_{\Sigma}(\mathcal{X})$.

Example 4. Let us consider the problematic CTRS in example 3. Its corresponding TRS generated as above contains the following rules:

$$\begin{array}{l}
\{\{x\}\} \to \{x\} \\
f(\{x\}, b) \to \{f(x, \mathit{true})\} \\
g(\{x\}) \to \{g(x)\} \\
\\
f(g(x), \mathit{true}) \to \mathit{if}(\mathit{equal?}(\{x\},\{0\}),\ \{x\},\ f(g(x), \mathit{false})), \\
g(g(x)) \to \{g(x)\}.
\end{array}$$

Then the term $f(g(g(0)), \mathit{true})$ admits the normal form $\{0\}$:

$$\begin{array}{rl}
f(g(g(0)), \mathit{true}) \to & \mathit{if}(\mathit{equal?}(\{g(0)\},\{0\}),\ \{g(0)\},\ f(g(g(0)), \mathit{false})) \to^* \\
& \to^* f(g(g(0)), \mathit{false}) \to f(\{g(0)\}, \mathit{false}) \to \\
& \to \{f(g(0), \mathit{true})\} \to \{0\}.
\end{array}$$

Note that the normal form $\{0\}$ is possible exactly because the information that a subterm has been rewritten is transmitted upwards via the operator $\{\_\}$

and its associated rules. The obtained TRS is not confluent, because the term above also admits the normal form 0, but this time the (at most two) normal forms that a term can have (t and/or $\{t\}$) are very closely related and one can easily infer the desired normal form in the original CTRS. In order to have a unique normal form in the TRS, we will actually enclose the original term into curly brackets before we reduce it, as the theorem below suggests.

Theorem 1. *If $\mathcal{R}$ is a conditional Σ-rewriting system then*

1. *For any ground Σ-terms α and β, $\alpha \to_{\mathcal{R}}^{\star} \beta$ if and only if there is some ground $\overline{\Sigma}$-term γ such that $\widehat{\gamma} = \beta$ and $\{\overline{\alpha}\} \to_{\overline{\mathcal{R}}}^{\star} \{\gamma\}$;*
2. *$\mathcal{R}$ terminates on a ground Σ-term α if and only if $\overline{\mathcal{R}}$ terminates on $\{\overline{\alpha}\}$;*
3. *$\overline{\mathcal{R}}$ terminates if and only if it terminates on all terms $\{\overline{\alpha}\}$ with α a Σ-term;*
4. *If γ in 1. is a normal form (in $\overline{\mathcal{R}}$) then β is also a normal form (in $\mathcal{R}$);*
5. *If $\mathcal{R}$ terminates then $\mathcal{R}$ is ground confluent iff $\overline{\mathcal{R}}$ is ground confluent.*

5 Putting Them All Together

We can now present the main result of this paper, namely a technique providing a rewriting engine that accepts conditional rewrite rules, obtained by appropriately wrapping a simpler rewriting engine that only accepts unconditional rules.

> ***Input:*** a conditional Σ-rewrite system $\mathcal{R}$ and a Σ-term α over variables X to be reduced with $\mathcal{R}$.
> ***Step 1:*** Add the variables in X as fresh constants into Σ, so that α becomes a ground Σ-term;
> ***Step 2:*** Generate $\overline{\Sigma}$ like in Subsection 4.2, by adding to each operator $\sigma \in \Sigma$ as many auxiliary arguments as conditional rules of lhs rooted in σ are in $\mathcal{R}$;
> ***Step 3:*** Generate the infrastructure unconditional $\overline{\Sigma}'$-rewrite system $\mathcal{I}(\overline{\Sigma})$ by adding to $\overline{\Sigma}$ the operators *equal*? and *if*$(\_,\_,\_)$ as well as their corresponding rules described in Section 3;
> ***Step 4:*** Generate the unconditional $\overline{\Sigma}'$-rewrite system $\overline{\mathcal{R}}$ by adding to $\mathcal{I}(\overline{\Sigma})$ the operation $\{\_\}$ and its rules, as well as the unconditional $\overline{\Sigma}'$-rewrite rules associated to the rules in $\mathcal{R}$ as shown in Subsection 4.4;
> ***Step 5:*** Reduce the term $\{\overline{\alpha}\}$ to a normal form in $\overline{\mathcal{R}}$, say $\{\gamma\}$, using any engine for unconditional rewriting;
> ***Step 6:*** Return the Σ-term $\widehat{\gamma}$.

We claim that the steps above, by applying a series of simple and totally automatic syntactic transformations to the input conditional rewrite system and term to rewrite, yield a rewriting engine that accepts conditional rewrite systems.

Step 1 shows a usual way to reduce terms with variables: interpret the variables as constants. However, in our framework it is quite important to add these variables explicitly as constants early in the reduction process. This is because the equality operator will need to consider these constants as distinct

operations, so that it can add appropriate rules, including ones of the form "$equal?(a, \sigma(x_1, \ldots, x_n)) \to false$" for any such constant a and operation $\sigma \in \Sigma_n$.

Step 2 modifies the signature Σ into $\overline{\Sigma}$, by analyzing the rules in $\mathcal{R}$ and adding an appropriate number of arguments to operations in Σ. Since the constants added to Σ at Step 1 are fresh, no rule in $\mathcal{R}$ has them as lhs terms, so these constants will not be changed in $\overline{\Sigma}$. Note, however, that other constants in Σ may get translated into operations with several arguments.

Step 3 adds the auxiliary equality and conditional operators that are needed to translate the conditional rules into unconditional ones. Note, again, that the constants added at Step 1 will increase the number of rules for *equal*?.

Step 4 generates the unconditional rewrite system $\overline{\mathcal{R}}$, by adding exactly one unconditional rule per conditional or unconditional rule in $\mathcal{R}$. Once $\overline{\mathcal{R}}$ is available, any engine for unconditional rewriting can be used to reduce $\overline{\alpha}$ under $\overline{\mathcal{R}}$, as done in Step 5. Note that the normal form γ of $\overline{\alpha}$ in $\overline{\mathcal{R}}$, if it exists, is a $\overline{\Sigma}$-term. Therefore, since the hat function is defined as $\widehat{\cdot} : T_{\overline{\Sigma}}(\mathcal{X}) \to T_{\Sigma}(\mathcal{X})$, the term $\widehat{\gamma}$ returned at Step 6 is indeed a Σ-term, as the result of reducing α under $\mathcal{R}$ is expected to be.

Theorem 2. *The algorithm above, taking a conditional Σ-rewrite system $\mathcal{R}$ and a Σ-term α as input, terminates iff $\mathcal{R}$ terminates on α. If the algorithm terminates and outputs a Σ-term β, then β is a normal form of α in $\mathcal{R}$.*

Proof. The algorithm terminates if and only if its Step 5 terminates, because all the other steps are nothing but simple syntactic translations over a finite signature and number of rewrite rules. Step 5 terminates if and only if $\overline{\mathcal{R}}$ terminates on $\{\overline{\alpha}\}$. By 2 in Theorem 1, this is equivalent to saying that $\mathcal{R}$ terminates on α. Now suppose that the algorithm terminates and that it outputs a Σ-term β. By Steps 5 and 6 and the discussion preceding this theorem, this happens if and only if there is some $\overline{\Sigma}$-term γ such that $\{\overline{\alpha}\} \to^{\star}_{\overline{\mathcal{R}}} \{\gamma\}$ and $\widehat{\gamma} = \beta$, which, by 1 in Theorem 1, is equivalent to $\alpha \to^{\star}_{\mathcal{R}} \beta$. Since γ is a normal form in $\overline{\mathcal{R}}$, it follows by 4 in Theorem 1 that β is a normal form in $\mathcal{R}$.

6 Preliminary Experiments

As mentioned previously in the paper, our original purpose for translating conditional rewrite systems into unconditional ones was to ease the process of implementing rewriting engines, at the same time aiming at highly efficient implementations of rewriting based on a fast and simple *rewriting virtual machine*. To test the computational equivalence between conditional rewriting systems and their corresponding unconditional variants, we have applied the translation presented in Section 5 manually for the conditional rewrite system in Example 1, and performed some experiments on a 2.4GHz PC machine using two major rewriting engines: Maude [4] and Elan [3]. The results of these experiments are listed in Figure 1, in seconds; the "-" should be read "core dump".

It was an unexpected surprise to see that the unconditional variant was much faster than its corresponding conditional rewrite system: almost 3 times faster for

	Maude		Elan	
n	Conditional	Unconditional	Conditional	Unconditional
11	0.007	0.003	0.604	0.061
13	0.027	0.009	2.309	0.244
15	0.142	0.037	8.922	0.959
17	0.459	0.154	34.2	3.8
21	7.9	2.6	548.1	62.9
23	29.2	10.4	-	-
25	117.2	49.3	-	-
31	7431	2489	-	-

Fig. 1. Times in seconds to reduce $odd(n)$ using Maude and ELAN, using both the conditional rewrite system in Example 1 and its unconditional variant in Example 2.

Maude and 10 times for ELAN. Since the implementation details of these rewrite engines are not well documented, we do not know the exact reasons for which conditional rewriting is so slow in these systems in comparison to unconditional rewriting. However, the preliminary results in Figure 1 tell us that maintaining the control context required to backtrack through conditional rewrite rules is a non-trivial matter, and that the translation technique proposed in this paper can be perhaps implemented in these systems to bring immediate benefits to conditional rewriting.

7 Conclusion and Future Work

An automatic technique to transform a conditional term rewriting system into an unconditional one was presented, which preserves all the major properties. The technique consists of adding some key auxiliary arguments to certain operations, which are used to maintain the control context as data context.

Only first steps towards a generic transformation procedure for general conditional rewrite systems have been made here. We have not considered rewriting modulo axioms yet, such as associativity, commutativity and identity, but these will be considered soon. Also, the current technique can be improved: not all the auxiliary arguments added to operations seem to be necessary. Can we statically reduce their number?

Future work will also investigate how well analysis techniques for unconditional rewriting systems translate into corresponding analysis techniques for conditional ones via our transformation. One complication here is that some of the operations that we introduce require to be evaluated eagerly in some arguments. This is also a drawback of our technique if one wants to use it for theorem proving purposes. Can one find a transformation which imposes no evaluation strategies?

References

1. S. Antoy, B. Brassel, and M. Hanus. Conditional narrowing without conditions. In *5th ACM SIGPLAN international conference on Principles and practice of declaritive programming (PPDP'03)*, pages 20–31. ACM Press, 2003.
2. J. Bergstra and J. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.
3. P. Borovansky, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P. Moreau, C. Ringeissen, and M. Vittek. *ELAN: User Manual*, 2000. Loria, Nancy, France.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual*, 2003. `http://maude.cs.uiuc.edu/manual`.
5. N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.
6. R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, volume 6.
7. D. P. Friedman, C. T. Haynes, and M. Wand. *Essentials of programming languages*. MIT Press, 1992.
8. E. Giovannetti and C. Moiso. Notes on the elimination of conditions. In *1st International Workshop on Conditional Term Rewriting Systems (CTRS'87)*, volume 308 of *LNCS*, pages 91–97. Springer, 1987.
9. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000.
10. M. Hanus. The integration of functions into logic programming: From theory to practice. *The Journal of Logic Programming*, 19 & 20:583–628, 1994.
11. C. Hintermeier. How to transform canonical decreasing ctrss into equivalent canonical trss. In *4th International Workshop on Conditional and Typed Rewriting Systems (CTRS'94)*, volume 968 of *LNCS*, pages 186–205, 1994.
12. J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *2nd International Joint Conference on Automated Reasoning (IJCAR'04)*, LNCS, to appear, 2004.
13. A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Journal of Applicable Algebra in Eng., Communication and Computing*, 5:313–353, 1994.
14. E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.
15. J. C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993.
16. G. Roşu. From conditional to unconditional rewriting. Technical Report UIUCDCS-R-2004-2471, University of Illinois at Urbana-Champaign, Aug. 2004.
17. P. Viry. Elimination of conditions. *Journal of Symbolic Computation*, 28:381–401, Sept. 1999.
18. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer, May 2001.