A Fold/Unfold Transformation Framework for Rewrite Theories and its Application to CCT
Technical Report

Maria Alpuente
DSIC, Universidad Polit\'ecnica de Valencia, Spain
alpuente@dsic.upv.es

Demis Ballis
DIMI, University of Udine, Italy
demis@dimi.uniud.it

Michele Baggi
Moreno Falaschi
DSMI, University of Siena, Italy
{baggi,moreno.falaschi}@unisi.it

Abstract

Many transformation systems for program optimization, program synthesis, and program specialization are based on fold/unfold transformations. In this paper, we present a fold/unfold–based transformation framework for rewriting logic theories which is based on narrowing. For the best of our knowledge, this is the first fold/unfold transformation framework which allows one to deal with functions, rules, equations, sorts, and algebraic laws (such as commutativity and associativity). We provide correctness results for the transformation system w.r.t. the semantics of ground reducts. Moreover, we show how our transformation technique can be naturally applied to implement a Code Carrying Theory (CCT) system. CCT is an approach for securing delivery of code from a producer to a consumer where only a certificate (usually in the form of assertions and proofs) is transmitted from the producer to the consumer who can check its validity and then extract executable code from it. Within our framework, the certificate consists of a sequence of transformation steps which can be applied to a given consumer specification in order to automatically synthesize safe code in agreement with the original requirements. We also provide an implementation of the program transformation framework in the high–performance, rewriting logic language Maude which, by means of an experimental evaluation of the system, highlights the potentiality of our approach.

Categories and Subject Descriptors I.2.2 [Artificial Intelligence]: Automatic Programming—Program Transformation; G.4 [Mathematics of Computing]: Mathematical Software—Certification and testing

General Terms Languages, Performance

Keywords Rewriting Logic, Fold/Unfold Program Transformation, Code Carrying Theory

1. Introduction

Transforming programs automatically to optimize their efficiency is one of the most fascinating techniques for rule–based programming languages [33, 43]. One of the most extensively studied program transformation approaches is the so called fold/unfold transformation system [8, 9, 16] (also known as the rules+strategies approach [35]). In this approach, the goal of obtaining a correct and efficient program is achieved in two phases, which may be performed by different actors: the first phase consists in writing an initial, maybe inefficient, program whose correctness can be easily shown; the second phase consists in transforming the initial program in order to obtain a more efficient one. This is often done by constructing a sequence of equivalent programs —called transformation sequence and usually denoted by $R_0, \ldots, R_n$—where each program $R_i$ is obtained from the preceding ones $R_0, \ldots, R_i$ by using an elementary transformation rule. The essential rules are folding and unfolding, i.e., contraction and expansion of subexpressions of a program using the definitions of the program itself (or of a preceding one). Other rules which have been considered are, e.g., instantiation, definition introduction/elimination and abstraction (sometimes referred with different names). When performing program transformation we may end up with a final program which is equal to the initial one, since the folding rule is the inverse of the unfolding rule. Thus, during the transformation process, we need strategies which guide the application of the transformation rules and can allow one to derive programs with improved performance. Some popular transformation strategies which have been proposed in the literature are the composition and tupling strategies. The composition strategy [9] is used to avoid the construction of intermediate data structures that are produced by some function $g$ and consumed by another function $f$. For some class of programs the composition strategy can be applied automatically. The tupling strategy [9, 17] proceeds by grouping calls with common arguments together so that their results are computed simultaneously. Unfortunately, the tupling strategy is more involved than the composition strategy and can in general be obtained only semi–automatically (although for particular classes of programs the tupling strategy has been completely automated [11, 12]).

A lot of literature has been devoted to proving the correctness of fold/unfold systems w.r.t. the various semantics proposed for logic programs [6, 22, 25, 27, 34, 38], functional programs [36, 37], and functional logic programs [1]. Quite often, however, transformations may have to be carried out in contexts in which the function symbols satisfy certain equational axioms. For example, in rule–based languages such as ASF+SDF [4], Elan [5], OBJ [23], CafeOBJ [18], and Maude [15] some function symbols may be de-
Our contribution. faithfully represented. Besides sorts, rules, equations and algebraic logical framework fold/unfold–based transformation methodology in the framework of rewriting logic [31], a flexible and expressive logical framework in which a wide range of logics and models of computation can be faithfully represented. Besides sorts, rules, equations and algebraic laws (such as commutativity and associativity), rewriting logic also features a code extractor to the set of function-defining axioms to term rewriting and unification [21]), and is efficiently implemented in the functional programming language Maude [13, 15].

Our contribution. The main contributions of the paper can be summarized as follows:

- We propose the first fold/unfold framework in the literature that applies to rewriting logic theories [29] and prove its correctness. Our methodology considers the possibility of transforming the equation set and the rule set of a rewrite theory separately in a way the semantics of ground reducts is proved to be preserved. We employ narrowing in order to empower the unfold operation by calculating the instance of an existing rule to embed the unfolding rule automatically via unification. The auxiliary transformation rules adopted apart from fold and unfold are: definition introduction and elimination, and abstraction.

- We chose to apply our transformation framework to the problem of securing the transfer of code from a code producer to a code consumer. Among the many different solutions that have been proposed to tackle this problem, we adhere to Code Carrying Theory (CCT) [40, 41]—a program synthesis framework stemming from a pioneering work of Manna and Waldinger [28] in which a theorem proving approach is taken to synthesize correct code from theorems. In this approach, programs are expressed as Term Rewriting Systems (TRSs) [26], and are transformed according to a given program transformation template expressed as a TRS too. Such a template consists of program schemas for input and output programs and a set of equations that the input and output programs must validate to guarantee the correctness of the transformation. A library of templates that matches the structure of the programs is required, otherwise the transformation cannot be applied.

Plan of the paper. The rest of the paper is organized as follows. In Section 2, we recall some necessary notions about rewriting logic and the Maude language. Section 3 formalizes the fold/unfold transformation system for rewrite theories and proves its correctness. In Section 4, we describe how the program transformation system can be exploited to implement CCT. Section 5 presents the prototypical implementation of our transformation framework. In Section 6, we draw some conclusions and discuss future work. The proofs of all the results of the paper are given in Appendix A. Finally, Appendix B outlines a technique for obtaining coherent and consistent equational theories.

2. Preliminaries

We consider an order-sorted signature $\Sigma$, with a finite poset of sorts $(S, \leq)$. We assume an $S$-sorted family $X = \{X_i\}_{i < S}$ of disjoint variable sets. $T_{\Sigma}(X)$, and $T_{\Sigma}$, are the sets of terms and ground terms of sort $s$, respectively. We write $T_{\Sigma}(X)$ and $T_{\Sigma}$ for the corresponding term algebras. The set of variables occurring in a term $t$ is denoted by $\mathit{Var}(t)$. We write $\mathit{Var}_{\Sigma}$ for the list of syntactic objects $o_1, \ldots, o_n$.

A position $p$ in a term $t$ is represented by a sequence of natural numbers $(\Lambda$ denotes the empty sequence, i.e., the root position). Positions are ordered by the prefix ordering: $p \leq q$, if $\exists w$ such that $p.w = q$. Given a term $t$, we let $\mathit{Pos}(t)$ and $\mathit{NVPos}(t)$ respectively denote the set of positions and the set of non-variable positions of $t$ (i.e., positions where a variable does not occur). $t[p]$ denotes the subterm of $t$ at position $p$, and $t[s][p]$ denotes the result of replacing the subterm $t[p]$ by the term $s$.

A substitution $\sigma$ is a mapping from variables to sequences of natural numbers ($\Lambda$ denotes the empty sequence, i.e., the root position). Substitutions are ordered by the prefix ordering: $\sigma \leq \tau$, if $\exists w$ such that $p.w = q$. Given a term $t$, we let $\mathit{Sub}(t)$ and $\mathit{NVSub}(t)$ respectively denote the set of substitutions and the set of non-variable substitutions of $t$ (i.e., substitutions where a variable does not occur). $t[\sigma]$ denotes the resulting term after applying the substitution $\sigma$ to the term $t$.

2010/03/9
associativity, commutativity, and identity declared for the different defined functions. We assume $\Sigma$ can be always considered as the disjoint union $\Sigma = C \cup D$ of symbols $c \in C$, called constructors, and symbols $f \in D$, called defined functions, each one having a fixed arity, where $D = \{ f \mid f(\bar{t}) = r \in \Delta \}$ and $C = \Sigma - D$. Then $T(C, X)$ is the set of constructor terms. Given an equation $l = r$, terms $l$ and $r$ are called the left-hand side (or lhs) and the right-hand side (or rhs) of the equation, respectively, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.

The equations in an equational theory $E$ are considered as simplification rules by using them only in the left to right direction, so for any term $t$, by repeatedly applying the equations as simplification rules, we eventually reach a term to which no further equations apply. The result is called the canonical form of $t$ w.r.t. $E$. This is guaranteed by the fact that $E$ is required to be terminating and Church-Rosser [7]. The set of equations in $\Delta$ together with the axiomatic axioms of $B$ in an equational theory $E$ induce a congruence relation on the set of terms $T_\Sigma(X)$ which is usually denoted by $\equiv_E$. $E$ is a presentation or axiomatization of $\equiv_E$. In abuse of notation, we speak of the equational theory $E$ to denote the theory axiomatized by $E$. Given an equational theory $E$, we say that a substitution $\sigma$ is a $E$-unifier of two generic terms $t$ and $t'$ if $\sigma t$ and $t' \sigma$ are both reduced to the same canonical form modulo the equational theory (in symbols $\sigma t = \sigma t'$).

A (order-sorted) rewrite theory is a triple $R = (\Sigma, \Delta \cup B, R)$, where $\Sigma$ is the union $D_1 \cup D_2 \cup C_1 \cup C_2$ such that $D_1 \cap D_2 = \emptyset$, and $(D_1 \cup C_1, \Delta \cup B)$ is an order-sorted equational theory. $R$ is a set of rewrite rules of the form $l \rightarrow r$ such that $l$ does not contain any symbol of $D_1$, $D_2 = \{ f \mid f(\bar{t}) \rightarrow r \in R \}$, and $C_2$ is the set of constructor symbols used in $R$. Symbols in $D_2$ are called defined symbols as well as those in $D_1$, with the only difference that the former are defined in rewrite rules, while the latter in the equational theory. In an analog way, this is valid for sets $C_1$ and $C_2$. We omit $\Sigma$ when no confusion can arise. Given a rule $l \rightarrow r$, terms $l$ and $r$ are called the left-hand side (or lhs) and the right-hand side (or rhs) of the rule, respectively, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. An equation of the form $t = t'$ or a a rule of the form $t \rightarrow t'$ are said to be:

1. Non-erasing, if $\mathcal{V}ar(t) = \mathcal{V}ar(t')$.
2. Sort preserving, if for each substitution $\sigma$, we have $\sigma t \in T_\Sigma(X)_s$ if and only if $t' \sigma \in T_\Sigma(X)_s$.
3. Sort decreasing, if for each substitution $\sigma$, $t' \sigma \in T_\Sigma(X)_s$ implies $\sigma t \in T_\Sigma(X)_s$.
4. Left (or right) linear, if $t$ (resp. $t'$) is linear, i.e., no variable occurs in the term more than once. It is called linear if both $t$ and $t'$ are linear.

A set of equations/rules is said to be non-erasing, or sort decreasing, or sort preserving, or (left or right) linear, if each equation/rule in it is so. Two equations of the form $l_0 = r_0$ and $l_1 = r_1$ overlap if there is a position $p_0 \in NVP\mathcal{P}(l_0)$ and substitution $\sigma$ such that $l_0|_p \sigma = b_1 l_1 \sigma$ or, viceversa, a position $p_1 \in NVP\mathcal{P}(l_1)$ and substitution $\sigma$ such that $l_1|_p \sigma = b_2 \sigma l_0$. A set of equations is said to be non-overlapping if there is no pair of overlapping rules. The same property can be easily lifted to rewrite rules.

We define the one-step rewrite relation on $T_\Sigma(X)$ as follows: $t \rightarrow_R t'$ if there is a position $p \in \mathcal{P}(l_0)$, a rule $l \rightarrow r$ in $R$, and a substitution $\sigma$ such that $l|_p \sigma = l\sigma$ and $t' = t[\sigma|p]$. An instance $l\sigma$ of a rule $l \rightarrow r$ is called a redex. A term $t$ without redexes is called normal form. A rewrite theory $R$ is weakly normalizing if every term has a normal form in $R$, though infinite rewrite sequences from $t$ may exist. The relation $\rightarrow_R$ for rewriting modulo $E$ is defined as $\equiv_E \circ \rightarrow_R \circ \equiv_E$. Let $\leq \subseteq A \times A$ be a binary relation on a set $A$. We denote the transitive closure by $\rightarrow^*$, the reflexive and transitive closure by $\rightarrow^*$, and the rewrite up to normal form by $\rightarrow^*$.

A rewrite theory is sufficiently complete if enough rules/equations have been specified, so that functions of the theory are fully defined on all relevant data. In the order sorted context, due to the sub-sort relation and overloading, sufficient completeness for weakly-normalizing, confluent and sort decreasing rewrite theories, can be checked by showing that for each term $t_1, \ldots, t_n$ where $f : s_1 \ldots s_n \rightarrow s$ is a defined symbol and every $t_i$ is a constructor term of sort $s_i$, $f(t_1, \ldots, t_n)$ is reducible in $R \cup E$ [24].

**Example 2.1.** Consider the following rewrite theory $(\Sigma, \Delta \cup B, R)$ such that $C_1 = \{ b, c \}$, $C_2 = \{ b, c, d \}$, $D_1 = \{ a, d \}$, $D_2 = \{ f \}$, $\Delta = \{ a = b, d = e \}$, $R = \{ f(b, c) \rightarrow d \}$ where $B$ contains the commutativity axiom for $f$. Then we can $R/E$-rewrite term $f(c, a)$ to $e$ by means of the following $R/E$ rewrite sequence $f(c, a) = \Delta f(c, b) = B f(b, c) \rightarrow_R d = \Delta e$.

### 3. Transforming Rewrite Theories

In this section, we introduce the narrowing-based transformation rules over rewrite theories and establish the correctness of the transformation system. We divide the transformation process into two steps. At the first step, we disregard of the rewrite rules and we only transform the set of equations $\Delta$ of the equational theory modulo the set of equational axioms $B$ (which are left unchanged). Then, we consider a new rewrite theory made of the transformed equational theory and the original rewrite rules. At the second step, we transform the rules modulo the new equational theory. This two-step process allows one to transform the rewrite rules modulo a fixed, already optimized, equational theory, which cannot change during the transformation of the rewrite rules. This fact results to be particularly helpful in proving the soundness of the whole fold/unfold framework.

#### 3.1 Narrowing in Rewriting Logic

Considering the rewrite relation $\rightarrow_{R/E}$, since $E$-congruence classes can be infinite, $\rightarrow_{R/E}$- reducibility is undecidable in general. One way to overcome this problem is to implement $R/E$-rewriting by a combination of rewriting using oriented equations and rules [42].

We adopt this approach in this paper.

We assume the following properties on $E = \Delta \cup B$.

(i) $B$ is non-erasing, and sort preserving.

(ii) $B$ has a finitary and complete unification algorithm, which implies that $B$-matching is decidable, and $\Delta \cup B$ has a complete (but not necessarily finite) unification algorithm.

(iii) $\Delta$ is sort decreasing, and confluent and terminating modulo $B$.

We define the relation $\rightarrow_{\Delta, B}$ on $T_\Sigma(X)$ as follows: $t \rightarrow_{\Delta, B} t'$ if there is a position $p \in \mathcal{P}(l_0)$, a rule $l \rightarrow r$ in $R$, and a substitution $\sigma$ such that $l|_p \sigma = l\sigma$ and $t' = t[\sigma|p]$. An instance $l\sigma$ of a rule $l \rightarrow r$ is called a redex. A term $t$ without redexes is called normal form. A rewrite theory $R$ is weakly normalizing if every term has a normal form in $R$, though infinite rewrite sequences from $t$ may exist. The relation $\rightarrow_{R/E}$ for rewriting modulo $E$ is defined as $\equiv_E \circ \rightarrow_{R} \circ \equiv_E$. Let $\leq \subseteq A \times A$ be a binary relation on a set $A$. We denote the transitive closure by $\rightarrow^*$, the reflexive and transitive closure by $\rightarrow^*$, and the rewrite up to normal form by $\rightarrow^*$.

(iv) $\rightarrow_{\Delta, B}$ is coherent with $B$, i.e., $\forall_{t_1, t_2, t_3}$, we have that $t_1 \rightarrow_{\Delta, B} t_2$ and that $t_1 =_{B} t_3$ implies $\exists_{t_4, t_5}$ such that $t_2 \rightarrow_{\Delta, B} t_4$, $t_3 \rightarrow_{\Delta, B} t_5$, and $t_4 =_{B} t_5$.

(v) $\rightarrow_{R, B}$ is $E$-consistent with $B$, i.e., $\forall_{t_1, t_2, t_3}$, we have that $t_1 \rightarrow_{R, B} t_2$ and that $t_1 =_{B} t_3$ implies $\exists_{t_4}$ such that $t_3 \rightarrow_{R, B} t_4$, and $t_2 =_{E} t_4$.
Every defined function symbol occurring in (1) non–deterministically reduce a term. The narrowing relation for (2) instead of matching in order to (3) and by performing uni–

Moreover, to ensure narrowing completeness we assume that (4) the right linearity of rules is needed to (5) the creation of equations/rules with extra-variables when performing folding steps. Consider, for instance, the folding of equation (6) and (7) an illegal equation.

3.2 Transformation rules

Let us define a FUN (Fold/Unfold transfor) as a rewrite theory satisfying conditions (i) – (viii) together with (ix) sufficient completeness, which is required for the completeness of the unfold transformation, as shown below. A FUN transformation sequence of length \( k \) for a rewrite theory \((\Sigma, \Delta \cup B, R)\) is a sequence \( (\mathcal{R}_0, \ldots, \mathcal{R}_k) \), \( k \geq 0 \), where each \( \mathcal{R}_i \) is a FUN theory, such that

- \( \mathcal{R}_0 = (\Sigma, E_0, R_0) \), with \( E_0 = (\Delta \cup B) \) and \( R_0 = R \).
- For each \( 0 \leq j < i \), \( \mathcal{R}_{j+1} = (\Sigma, \Delta_{j+1} \cup B, R_0) \) is derived from \( \mathcal{R}_j \) by an application of a transformation rule on the equation set \( \Delta_j \).
- For each \( i \leq j < k \), \( \mathcal{R}_{j+1} = (\Sigma, E_i, R_{j+1}) \) is derived from \( \mathcal{R}_j \) by an application of a transformation rule on the rule set \( R_j \).

The transformation rules are introduction, elimination, fold, unfold, and abstraction, which are defined as follows.

Definition Introduction. We can obtain program \( \mathcal{R}_{k+1} \) by adding to \( \mathcal{R}_k \) a set of new equations (resp. rules), defining a new symbol \( f \) called eureka. We consider equations (resp. rules) of the form \( f(\overline{t}) = r_i \) (resp. \( f(\overline{t}) \to r_i \)), such that:

1. \( f \) is a function symbol which does not occur in the sequence \( \mathcal{R}_0, \ldots, \mathcal{R}_k \) and is declared by \( f : s_1 \ldots s_n \to s \) where \( s_1, \ldots, s_n, s \) are sorts declared in \( \mathcal{R}_0 \) and \( \overline{x} \) are equational attributes.
2. \( t_i \in T (\mathcal{C}, \mathcal{X}) \), and \( \forall \text{var}(\overline{t}) = \forall \text{var}(r_i) \), for all \( i \) – i.e., the equations/rules are non-erasing.
3. Every defined function symbol occurring in \( r_i \) belongs to \( \mathcal{R}_0 \).
4. The set of new equations (resp. rules) are left linear, sufficient complete and non overlapping. For rules we require also right linearity.

In general, the main idea consists of introducing new auxiliary function symbols which are defined by means of a set of equations/rules whose bodies contain a subset of the functions that appear in the right-hand side of an equation/rule that appears in \( \mathcal{R}_0 \), whose definition is intended to be improved by subsequent transformation steps. The non overlapping property and the left linearity ensure confluence of eurekas, which is needed to preserve the completeness of the fold operation and will be discussed later. Sufficient narrowing is needed to ensure the completeness of unfolding and will be discussed later. Right linearity on rules is needed to ensure narrowing completeness [30], and left linearity is also needed to preserve the right linearity of rules when doing folding. Consider, for instance, the folding of rule \( f(x) = g(x) \) using the (non left linear) eureka \( new(x,x) = g(x) \), which would produce a new rule \( f(x) = new(x,x) \) which is not right linear.

Note that, once a transformation is applied to a eureka, the obtained equation/rule is not considered to be a eureka anymore. As we will see later, this is important for the folding operation, since we can only fold non-eureka equations/rules using eureka ones. The non-erasing condition is a standard requirement that avoids the creation of equations/rules with extra-variables when performing folding steps. Consider, for instance, the folding of equation \( f(x) = g(x) \) using the (erasing) eureka \( new(x,y) = g(x) \), which would produce a new equation \( f(x) = new(x,y) \) containing an extra variable in its right-hand side (thus an illegal equation).

Definition Elimination. Let \( \mathcal{R}_k \) be the rewrite theory \((\Sigma_k, \Delta_k \cup B_k, R_k)\). We can obtain program \( \mathcal{R}_{k+1} \) by deleting from program \( \mathcal{R}_k \):

- \( \forall \text{all equations that define the functions } f_0, \ldots, f_n \), say \( \Delta_k' \), such that \( f_0, \ldots, f_n \) do not occur either in \( \mathcal{R}_0 \) or in \( (\Sigma_k \setminus \Delta_k') \cup B_k, \mathcal{R}_k \).
- \( \forall \text{all rules that define the functions } f_0, \ldots, f_n \), say \( R_k' \), such that \( f_0, \ldots, f_n \) do not occur either in \( \mathcal{R}_0 \) or in \( (\Sigma_k \setminus \Delta_k' \cup B_k, \mathcal{R}_k \setminus R_k') \).

Note that the deletion of the equations/rules that define a function \( f \) implies that no function calls to \( f \) are allowed afterwards. However, subsequent transformation steps (in particular, folding steps) might introduce those deleted functions in the rhs’s of the equations/rules, thus producing inconsistencies in the resulting programs. To avoid this, we forbid any folding step after a definition elimination has been performed (this generally boils down to postpone all elimination steps to the end of the transformation sequence).

Unfolding. Let \( F \in \mathcal{R}_k \) be an equation of the form \( l = r \). We can obtain program \( \mathcal{R}_{k+1} \) from program \( \mathcal{R}_k \) by replacing \( F \) with the set of equations

\[ \{ \sigma \rightarrow \rho \mid \sigma \sim_{\mathcal{R}_k} \Delta_k, \mathcal{B}_k \} \]

Note that the narrowing steps are performed by using only the equations implicitly oriented as rewrite rules.

Let \( F \in \mathcal{R}_k \) be a rule of the form \( l \rightarrow r \). We say that another rule \( R' \rightarrow r' \in \mathcal{R}_k \) is evil for \( F \), if there exist a substitution \( \rho \) and position \( p \in N \forall \text{phot}(l') \), such that \( R' \rightarrow_{\rho} l \). If in \( \mathcal{R}_k \) there is no evil rule for \( F \), then we can obtain program \( \mathcal{R}_{k+1} \) from program \( \mathcal{R}_k \) by replacing \( F \) with the set of rules

\[ \{ l \sigma \rightarrow r' \mid \sigma \sim_{\mathcal{R}_k} \Delta_k, \mathcal{B}_k \} \]

The specified property of evil rules makes them unusable to unfold the selected rule because they do not allow a narrowing step from \( r \). The evil rule constraint is needed to guarantee the correctness of unfolding as shown in the following example.
Example 3.1. Consider the following rewrite theory $\mathcal{R} = (\Sigma_\mathcal{R}, \emptyset, R)$, where $\Sigma_\mathcal{R}$ is the signature containing all the symbols of $\mathcal{R}$ and

$$
R : 
\begin{align*}
g_1(x) &\rightarrow x \\
h_1(0) &\rightarrow 0 \\
h_1(1) &\rightarrow 0 \\
h(s(x)) &\rightarrow 0
\end{align*}
$$

and

$$
R' : 
\begin{align*}
g_1(x) &\rightarrow x \\
h_1(0) &\rightarrow 0 \\
h_1(1) &\rightarrow 0 \\
h(s(x)) &\rightarrow s(x) \\
g_2(x) &\rightarrow s_1(g_1(x)) \\
h(s_1(g_1(x))) &\rightarrow 1
\end{align*}
$$

We get program $R' = (\Sigma_\mathcal{R}, \emptyset, R')$ from $\mathcal{R}$ by applying an unfolding step over the rule defining function $g_2$ in $\mathcal{R}$, performing the narrowing step $g_2(x) \leadsto s(x)$. Term $h(g_2(0))$ can be rewritten in $\mathcal{R}$ to the normal forms 0 or 1 by means of the rewrite sequences $h(g_2(0)) \rightarrow h(s_0(0)) \rightarrow h(0) \rightarrow 0$, and $h(g_2(0)) \rightarrow h(s_1(0)) \rightarrow 1$. The only possible rewrite sequence from $h(g_2(0))$ in $\mathcal{R}'$ is $h(s_1(0)) \rightarrow 0$, missing solution 1. This is due to the rule $h(s_1(g_1(x))) \rightarrow 1$ which is evil for $g_2(x) \rightarrow s_1(g_1(x))$ because the rhs $s_1(g_1(x))$ is embedded in the lhs $h(s_1(g_1(x)))$.

The use of narrowing empowers the unfold operation by implicitly embedding the instantiation rule (the operation of the Burstall and Darlington framework) that introduces an instance of an existing rule into unfolding by means of unification. It is worth noting that the rewrite theory has to be sufficiently complete in order to preserve the completeness of unfolding, as witnessed by the following example.

Example 3.2. Consider the following rewrite theory $\mathcal{R} = (\Sigma_\mathcal{R}, \emptyset, R)$, where $\Sigma_\mathcal{R}$ is the signature containing all the symbols of $\mathcal{R}$ and

$$
R = \left\{ \begin{array}{c}
f(0) \rightarrow 0 \\
g(x) \rightarrow s(f(x)) \\
h(0) \rightarrow 0 \\
h(s(x)) \rightarrow s(0)
\end{array} \right\}
$$

and

$$
R' = \left\{ \begin{array}{c}
f(0) \rightarrow 0 \\
g(0) \rightarrow s(0) \\
h(0) \rightarrow 0 \\
h(s(x)) \rightarrow s(0)
\end{array} \right\}
$$

We get program $R' = (\Sigma_\mathcal{R}, \emptyset, R')$ from $\mathcal{R}$ by applying an unfolding step over the rule defining function $g$ in $\mathcal{R}$, performing the narrowing step $s(f(x)) \leadsto s(0)$. Now, the term $h(g(s(0)))$ can be rewritten in $\mathcal{R}$ to the normal form $s(0)$, whereas this is no longer possible in the transformed program. This is due to the non-sufficient completeness of $\mathcal{R}$, in particular to the partial definition of $f$ so that the term $f(s(0))$ is not reducible in $\mathcal{R}$. Hence, by unfolding the call $f(x)$, we improperly impose an unnecessary restriction in the domain of the function $g$.

Folding. Let $F \in \mathcal{R}_k$ be an equation (the "folded equation") of the form $(l = r)$, and let $F' \in \mathcal{R}_l$, $0 \leq j \leq k$, be an equation (the "folding equation") of the form $(l' = r')$, such that $r_j = b_k \ r' \sigma$ for some position $p \in N\forall Pos(r)$ and substitution $\sigma$. Note that, since we transform the equations of an equational theory, we consider here the congruence relation $\equiv_B$ modulo the equational axioms $B_\mathcal{R}$ (assuming an empty equation set). This is because we cannot consider a congruence modulo an equational theory which is being modified. Moreover, the following conditions must be satisfied:

1. $F$ is not a eureka.
2. $F'$ is a eureka.
3. The substitution $\sigma$ is sort decreasing, i.e. if $x \in X_s$, then $s(x) \in \Sigma_\mathcal{R}(X_s)$, such that $s' \leq s$.
4. Let $l' = f(l_{\sigma})$ and $r' = e$ and let $f(l_{\sigma})$ and $e$ have type $s_f$ and $s_e$, respectively; then $s_f \leq s_e$.

Then, we can obtain program $\mathcal{R}_{k+1}$ from program $\mathcal{R}_k$ by replacing $F$ with the new equation $(l = r'[\sigma])$.

Folding can be applied to rules whenever the transformation of the equational theory has been completed. To fold rules we proceed as follows. Let $F \in \mathcal{R}_k$ be a rule (the "folded rule") of the form $(l = r)$, and let $F' \in \mathcal{R}_j$, $0 \leq j \leq k$, be a rule (the "folding rule") of the form $(l' = r')$, such that $r_j' = E_k \ r' \sigma$ for some position $p \in N\forall Pos(r)$ and substitution $\sigma$, fulfilling conditions (1) - (4) above. Then, we can obtain program $\mathcal{R}_{k+1}$ from program $\mathcal{R}_k$ by replacing $F$ with the new rule $(l = r'[\sigma])$. Note that in this case we use the congruence modulo the equational theory $E_k$ since it does not change any more after this stage.

The need for conditions (1) and (2) is twofold. These conditions forbid self-folding, that is, a folding operation with $F = F'$, thus a rule with the same left and right-hand side cannot be produced, which may introduce infinite loops on derivations and destroy the correctness properties of the transformation system. These conditions also forbid the folding of a eureka, which is meaningless as illustrated in the following example.

Example 3.3. Consider the following two rules:

\[
\begin{array}{c}
new \rightarrow f \\
g \rightarrow f
\end{array}
\]

Without conditions (1) and (2), a folding of the eureka rule would be possible, obtaining the new rule (new $\rightarrow g$), which is nothing more than a redefinition of the symbol new. Since transformation rules aim to optimize the original program with the support of eurekas, a folding over a eureka is meaningless or even dangerous.

Finally, conditions (3) and (4) ensure the sort compatibility of both the applied substitutions and the term that is inserted into the folded equation/rule right-hand side.

When presenting the definition introduction operation, we said that eurekas have to be confluent in order to ensure the completeness of the fold operation. We now discuss this point by means of an example.

Example 3.4. Consider the following rewrite theory $\mathcal{R} = (\Sigma_\mathcal{R}, \emptyset, R)$, where $\Sigma_\mathcal{R}$ is the signature containing all the symbols of $\mathcal{R}$ and

$$
R : 
\begin{align*}
f(a,b) &\rightarrow g(a,b) \\
f(x,y) &\rightarrow g(x,y) \\
m(a) &\rightarrow a \\
m(b) &\rightarrow b \\
g(a,x) &\rightarrow a \\
g(b,x) &\rightarrow b
\end{align*}
$$

and

$$
R' : 
\begin{align*}
f(a,b) &\rightarrow g(m(a), b) \\
f(x,y) &\rightarrow g(x,y) \\
m(a) &\rightarrow m(a) \\
m(b) &\rightarrow m(b) \\
g(a,x) &\rightarrow g(a,x) \\
g(b,x) &\rightarrow g(b,x)
\end{align*}
$$

We get program $\mathcal{R}' = (\Sigma_\mathcal{R}, \emptyset, R')$ from $\mathcal{R}$ by applying a folding step to the rule $f(a,b) \rightarrow g(a,b)$ using the eureka $m(a) \rightarrow a$. It is easy to see that in $\mathcal{R}'$ we can reduce term $f(a,b)$ to the normal forms $a$ or $b$, whereas in $\mathcal{R}$ we can reach only the normal form $a$. The point is that in $\mathcal{R}$, term $f(a,b)$ can reduce only to $g(a,b)$ while the fold operation introduces the possibility of rewriting to $g(b,x)$ cause the eureka defining $m$ is not confluent. This leads to a new solution $b$, thus missing the completeness.

Abstraction. The set of rules presented so far constitutes the core of our transformation system; however let us mention another useful rule, called abstraction, which can be simulated in our settings by applying appropriate definition introduction and folding steps. This rule is usually required to implement tupling, and it consists of replacing, by a new function, multiple occurrences of the same expression $e$ in the right-hand side of an equation/rule. For instance,
consider the following equation
\[ double\_sum(x, y) = sum(sum(x, y), sum(x, y)) \]
where \( e = sum(x, y) \). The equation can be transformed into the following pair of equations
\[
\begin{align*}
\text{double}_\text{sum}(x, y) &= \text{ds}_\text{aux}(\text{sum}(x, y)) \\
d\text{ds}_\text{aux}(z) &= \text{sum}(z, z)
\end{align*}
\]
These equations are generated from the original one by a definition introduction of the eureka \( \text{ds}_\text{aux} \) and then by folding the original equation by means of the newly generated eureka.

Note that the abstraction rule applies on equations or rules which are not right linear, since the same expression \( e \) occurs more than once in their rhs. Since we ask for rules to be right linear for the completeness of the narrowing relation, we may think to use the abstraction rule to preprocess rewrite rules in order to try to make them right linear.

### 3.3 Correctness of the transformation system

Given a rewrite theory \( \mathcal{R} \), let us define the set \( \text{red}_{\mathcal{R}} = \{s \mid \exists t \in \mathcal{T}_{\mathcal{R}} \land t \rightarrow^* R s \} \), as the set of all ground terms reachable in \( \mathcal{R} \) in a finite number of rewrite steps (also called ground reducts). Let also be \( \text{GNF}_{\mathcal{R}} = \{s \mid \exists t \in \mathcal{T}_{\mathcal{R}} \land t \rightarrow^* R s \} \subseteq \text{red}_{\mathcal{R}} \) the set of all ground normal forms reachable in \( \mathcal{R} \).

Theorem 3.1 states the main theoretical result for the transformation system based on the elementary rules introduced so far: definition introduction, definition elimination, unfolding, folding, and abstraction. The result is strong correctness of a transformation sequence, i.e., the semantics of the ground reducts is preserved modulo the equational theory as stated by Theorem 3.1. The proof of the Theorem can be found in Appendix A.

**Theorem 3.1.** Let \( (\mathcal{R}_0, \ldots, \mathcal{R}_k), k > 0 \), be a FUN transformation sequence. Then, \( \text{GNF}_{\mathcal{E}_0} =_{R} \text{GNF}_{\mathcal{E}_k} \), \( \text{GNF}_{\mathcal{R}_0} =_{\mathcal{E}_0} \text{GNF}_{\mathcal{R}_k} \), and for all \( t \in \mathcal{T}_{\mathcal{R}_0} \), if \( s \in \text{red}_{\mathcal{R}_0}(t) \) then there exist \( s_1, s_2 \) such that \( s_1 \in \text{GNF}_{\mathcal{R}_k}(t) \), \( s_2 \in \text{GNF}_{\mathcal{R}_k}(s) \), and \( s_1 =_{\mathcal{E}_0} s_2 \). Viceversa, for all \( t \in \mathcal{T}_{\mathcal{R}_0} \), if \( s \in \text{red}_{\mathcal{R}_k}(t) \) then there exist \( s_1, s_2 \) such that \( s_1 \in \text{GNF}_{\mathcal{R}_k}(t) \), \( s_2 \in \text{GNF}_{\mathcal{R}_k}(s) \), and \( s_1 =_{\mathcal{E}_0} s_2 \).

The following example demonstrates that the theorem above cannot be lifted to the non-ground semantics of reducts.

**Example 3.5.** Consider the FUN theory \( \mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R) \) where

\[
R = \left\{ \begin{array}{ll}
  f(0) & \rightarrow 0 \\
  f(s(x)) & \rightarrow s(x) \\
  g(x) & \rightarrow s(f(x))
\end{array} \right\}
\]

We get the rewrite theory \( \mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R') \) from \( \mathcal{R} \) by applying an unfolding step over the rule defining function \( g \) in \( R \). Then, consider the non-ground term \( g(x) \). In \( \mathcal{R} \), we have a (one step) derivation from term \( g(x) \) to the normal form \( s(f(x)) \), whereas in \( \mathcal{R}' \) there is no derivation starting from term \( g(x) \). So, the reduct \( s(f(x)) \) is not preserved by the transformation. The same example also shows that not even a more restricted non-ground semantics, such as the non-ground normal form semantics, is preserved. Nevertheless, in the reachability context of rewrite theories where confluence or termination are not required, this semantics is neither reasonable nor useful.

### 4. CCT via Program Transformation

In this section, we explain how the transformation system presented so far can be employed to implement our CCT approach. The CCT methodology consists of several steps, which are illustrated in Figure 4, and summarized below.

1. **Defining Requirements** (Code Consumer). The code consumer provides the requirements to the code producer in the form of a rewrite theory, specifying the functions of interest with a naïve, non-optimized, even redundant piece of code. The rewrite theory can be written in Maude [14], a high-level specification language that implements rewriting logic [29].

2. **Defining New Functions** (Code Producer). The code producer has to generate an efficient implementation of the specified functions and a proof that such an implementation satisfies the required specifications. To this aim, the code producer uses the fold/unfold-based transformation system presented in Section 3 to (semi-)automatically obtain an efficient implementation of the specified functions. Moreover, some specific strategies such as composition and tupling can be easily automated (see [1] for more details). Subsequently, rather than sending the efficient functions as actual code to the consumer, the producer will send only a certificate consisting of a compact representation of the transformation rule sequence employed to derive the program. The strong correctness of the transformation system ensures that the obtained program is correct w.r.t. the initial consumer specifications, so the code producer does not need to provide extra proofs.

3. **Code Extraction** (Code Consumer). Assuming the transformation infrastructure is publicly available, once the certificate is received, the code consumer can apply the transformation sequence, described in the certificate, to the requirements, and the final program can be obtained without the need of other auxiliary software for the code extraction.

Definition 4.2 formalizes the notion of certificate for a transformation sequence. In order to build a certificate, we need a way to describe a transformation rule, which is achieved by a transformation rule description.

**Definition 4.1.** We associate a transformation rule description with each transformation rule, as follows:

- **Definition Introduction Description:**
  - Intro(Operator Declaration, Equation Set)
  - Intro(Operator Declaration, Rule Set)
- **Elimination Description:**
  - Elim(List of function symbols)
- **Unfolding Description:**
  - Unfold(Unfolded equation id, Unfold position)
  - Unfold(Unfolded rule id, Unfold position)
- **Folding Description:**
  - Fold(Folding equation id, Folded equation id, Fold position)
  - Fold(Folding rule id, Folded rule id, Fold position)

Note that rules and equations are referenced by an identification label which can be systematically generated and assigned to each rule/equation. We assume that the identification label for equations (resp. rewrite rules) is of the form \( E_n \) (resp. \( R_n \)), where \( n \) is a progressive number. More specifically, when a transformation rule is applied to a given rewrite theory and a new equation (resp. rule) is produced, a fresh identification label \( E_n \) (resp. \( R_n \)), is created and associated with the corresponding rule/equation.

It is also worth noting that rule/equation descriptions can precisely identify terms to be folded/unfolded by using the standard notation for term positions.

**Definition 4.2** (Certificate). Let \( (\mathcal{R}_0, \ldots, \mathcal{R}_k), k > 0 \), be a transformation sequence. The certificate associated with the transformation sequence \( (\mathcal{R}_0, \ldots, \mathcal{R}_k) \) is the ordered list of transfor-
mation rule descriptions \(d_1, \ldots, d_k\) associated with the transformation rules \(r_1, \ldots, r_k\) s.t. \(\forall i \in \{1, \ldots, k\}, r_i\) is the transformation rule applied to \(R_{i-1}\) to obtain \(R_i\).

Let us show some selected examples to illustrate this.

**Example 4.1.** Let us now consider a simple specification of the Fibonacci function which uses the usual Peano notation to represent natural numbers. The specification is modeled by means of the following naive equational theory.

\[
\begin{align*}
\text{op } \text{fib} & : \text{Nat} \rightarrow \text{Nat} . \\
\text{(E1)} & \quad \text{eq } \text{fib}(0) = S(0) . \\
\text{(E2)} & \quad \text{eq } \text{fib}(S(0)) = S(0) . \\
\text{(E3)} & \quad \text{eq } \text{fib}(S(S(n))) = \text{fib}(S(n)) + \text{fib}(n) .
\end{align*}
\]

Due to the highly recursive nature of this definition of \(\text{fib}\), the evaluation of an expression like \(\text{fib}(S(S(0)))\) will compute many calls to the same instances of the function again and again, and it will expand the original term into a whole binary tree of additions before collapsing it to a number. The exponential number of repeated function calls makes the evaluation of \(\text{fib}(S^50(0))\) will compute many

(1) First, we introduce the following eureka which makes use of the tupling strategy.

\[
\begin{align*}
\text{sort } \text{Pair} . \\
\text{op } \langle ., . \rangle & : \text{Nat} \times \text{Nat} \rightarrow \text{Pair} . \\
\text{(E4)} & \quad \text{eq } \text{aux}(n) = \langle \text{fib}(S(n)), \text{fib}(n) \rangle .
\end{align*}
\]

(2) We now unfold the redex \(\text{fib}(S^5(0))\) of equation (E4).

\[
\begin{align*}
\text{(E5)} & \quad \text{eq } \text{aux}(0) = \langle S(0), \text{fib}(0) \rangle . \\
\text{(E6)} & \quad \text{eq } \text{aux}(S(n)) = \langle \text{fib}(S(n)) + \text{fib}(n), \text{fib}(S(n)) \rangle .
\end{align*}
\]

We unfold once again equation (E5) in order to remove the call to \(\text{fib}\).

\[
\text{(E7)} \quad \text{eq } \text{aux}(0) = \langle S(0), S(0) \rangle .
\]

(3) Then, abstraction is applied to equations (E3) and (E6) by means of two new eurekas

\[
\begin{align*}
\text{op } \text{aux2} & : \text{Pair} \rightarrow \text{Nat} . \\
\text{op } \text{aux3} & : \text{Pair} \rightarrow \text{Pair} . \\
\text{(E8)} & \quad \text{eq } \text{aux2}(x,y) = x + y . \\
\text{(E9)} & \quad \text{eq } \text{aux3}(x,y) = \langle x + y, x \rangle .
\end{align*}
\]

The second step for the abstraction is the folding of equations (E3) and (E6) by means of eurekas (E8) and (E9) respectively.

\[
\begin{align*}
\text{(E10)} & \quad \text{eq } \text{fib}(S(S(n))) = \text{aux2}((\text{fib}(S(n)), \text{fib}(n))) . \\
\text{(E11)} & \quad \text{eq } \text{aux}(S(n)) = \text{aux3}((\text{fib}(S(n)), \text{fib}(n))) .
\end{align*}
\]

(4) Finally, the right-hand sides of both equations are folded using the original definition of function \(\text{aux}\).

\[
\begin{align*}
\text{(E12)} & \quad \text{eq } \text{fib}(S(S(n))) = \text{aux2}(\text{aux}(n)) . \\
\text{(E13)} & \quad \text{eq } \text{aux}(S(n)) = \text{aux3}(\text{aux}(n)) .
\end{align*}
\]

The transformed (linear) definition of the equational theory for \(\text{fib}\) is as follows.

\[
\begin{align*}
\text{(E1)} & \quad \text{eq } \text{fib}(0) = S(0) . \\
\text{(E2)} & \quad \text{eq } \text{fib}(S(0)) = S(0) . \\
\text{(E12)} & \quad \text{eq } \text{fib}(S(S(n))) = \text{aux2}(\text{aux}(n)) . \\
\text{(E7)} & \quad \text{eq } \text{aux}(0) = (S(0), S(0)) . \\
\text{(E13)} & \quad \text{eq } \text{aux}(S(n)) = \text{aux3}(\text{aux}(n)) . \\
\text{(E8)} & \quad \text{eq } \text{aux2}(x,y) = x + y . \\
\text{(E9)} & \quad \text{eq } \text{aux3}(x,y) = \langle x + y, x \rangle .
\end{align*}
\]

The resulting certificate \(C\) is as follows.

\[
C = (\text{Intro}(\text{op } \text{aux} : \text{Nat} \rightarrow \text{Pair}), \\
(\text{eq } \text{aux}(n) = (\text{fib}(S(n)), \text{fib}(n))), \text{Unfold}(\text{E4}, [1]), \\
\text{Unfold}(\text{E5}, [2]), \text{Intro}(\text{op } \text{aux2} : \text{Pair} \rightarrow \text{Nat}), \\
(\text{eq } \text{aux2}(x,y) = x + y), \\
\text{Intro}(\text{op } \text{aux3} : \text{Pair} \rightarrow \text{Pair}), \\
(\text{eq } \text{aux3}(x,y) = \langle x + y, x \rangle), \text{Fold}(\text{E8}, \text{E3}, \text{A}), \\
\text{Fold}(\text{E9}, \text{E6}, \text{A}), \text{Fold}(\text{E4}, \text{E10}, [1]), \text{Fold}(\text{E4}, \text{E11}, [1])).
\]

**Example 4.2.** Suppose the code consumer needs a function for computing the sum of the Fibonacci values of the natural numbers in a list. The type of the list of natural numbers is predefined in Maude. The consumer specification is a rewrite theory which consists of the equational theory of Example 4.1 defining the Fibonacci function along with the following set of rules.

\[
\begin{align*}
\text{op } \text{sum-list} & : \text{NatList} \rightarrow \text{Nat} . \\
\text{(R1)} & \quad \text{rl } \text{sum-list}([nil]) \Rightarrow 0 . \\
\text{(R2)} & \quad \text{rl } \text{sum-list}(\text{xs}) \Rightarrow \text{fib}(\text{xs}) + \text{sum-list}(\text{xs}) .
\end{align*}
\]

The equational theory defining the Fibonacci function can be optimized as shown in Example 4.1. The above rules defining the
sum-list function can be transformed in a more efficient tail-
recursive structure by using our fold/unfold framework as follows.
(1) We first introduce the following definition

\[
\text{op sum-list-aux} : \text{NatList Nat} \rightarrow \text{Nat}.
\]

(R3) \[ \text{rl sum-list-aux}(x,x) \Rightarrow x + \text{sum-list}(x). \]

(2) By applying the unfold operation over the eureka (R3), we
obtain the following new rules

\[
\begin{align*}
(R4) & \quad \text{rl sum-list-aux}(\text{nil},x) \Rightarrow x. \\
(R5) & \quad \text{rl sum-list-aux}(y,ys,x) \Rightarrow x*y + \text{sum-list}(ys). \\
\end{align*}
\]

(3) Now, by folding rule (R2) and (R5) using the eureka (R3), we
obtain the final tail-recursive program.

\[
\begin{align*}
(R1) & \quad \text{rl sum-list}(\text{nil}) \Rightarrow 0. \\
(R6) & \quad \text{rl sum-list}(x,xs) \Rightarrow \text{sum-list-aux}(x,\text{fib}(x)). \\
(R4) & \quad \text{rl sum-list-aux}(\text{nil},x) \Rightarrow x. \\
(R7) & \quad \text{rl sum-list-aux}(x,ys,x,y) \Rightarrow \\
& \quad \text{sum-list-aux}(x,\{\text{fib}(x) + y\}). \\
\end{align*}
\]

The certificate \( \mathcal{C} \) is then as follows.

\[
\mathcal{C} = (\text{Intro}(\text{op sum-list-aux} : \text{NatList Nat} \rightarrow \text{Nat}), \\
\{\text{rl sum-list-aux}(x,xs) \Rightarrow x + \text{sum-list}(xs)\}). \\
\text{Unfold}(\text{R3}, [1,2]), \text{Fold}(\text{R3}, \text{R5}, \Lambda), \text{Fold}(\text{R3}, \text{R2}, \Lambda)).
\]

By applying now the certificate to the initial specification, the
code consumer can efficiently obtain the required efficient imple-
mentation.

5. Implementation

We implemented the transformation framework presented in Sec-
tion 3 in a prototypical system, which consists of about 500 lines of
code, written in Maude. Basically, our system allows us to perform
the elementary transformation rules over a given initial program.
The prototype also provides a simple user interface that shows a
menu with the available operations as well as the program resulting
after applying each transformation rule. Therefore, it is possible to
see the complete transformation sequence. A snapshot of the in-
terface is shown in Figure 5. In order to implement the transforma-

As already mentioned, this framework opens up new appli-
cations to program optimization, program synthesis, program spe-
cialization and theorem proving for first-order typed rule-based lan-
guages such as ASF+SDF, Elan, OBJ, CafeOBJ, and Maude.

We have shown that our methodology can be effectively applied
to CCT and may significantly simplify the code producer task.
Actually, by applying a strategy such as composition, tupling, etc., the
code producer can (semi-)automatically obtain the final improved
program. Of course, for the methodology to pay off in practice,
the transformation system could be instrumented to also provide
an estimation of the achieved optimization. However, this is out
of the scope of this paper. We intend to investigate such an exten-
sion in a future work. Thanks to the fact that our transformation
methodology relies on narrowing, this can be done by adapting
the narrowing-based transformation strategies of [1]. Moreover,
as an outcome of the transformation process, a compact representa-
tion of the sequence of applied transformation rules is delivered
as a certificate to the code consumer. The code consumer needs
only applying the certificate to the initial requirements to obtain
the desired program. This checking can be completely automated.
We provided a prototypical implementation of the transforma-
tion framework written in Maude. Some available Maude formal tools
can be employed to verify relevant program properties required by
our methodology such as termination [19] of the equation set, con-
We are now implementing the complete CCT infrastructure along
with several fully automatic transformation strategies on top of the
transformation system. Moreover, we are developing a number of
transformation templates in the sense of [10] for rewriting logic
theories, whose correctness should (hopefully) follow from our re-

References

for Transforming Lazy Functional Logic Programs. Theoretical


http://www.csail.mit.edu/~kostas/dpla/athena/
athenaTutorial.pdf.


[5] P. Borovanský, C. Kirchner, H. Kirchner, and P. E. Moreau. ELAN
from a Rewriting Logic Point of View. Theoretical Computer Science,

preserve Computed Answer Substitutions of Logic Programs. Journal

in Membership Equational Logic. Theoretical Computer Science,

The prototype is freely available along with some examples at
http://users.dimi.uniud.it/~michele.baggi/cct/

6. Conclusions

In this paper, we provide a novel rewriting logic framework for
Code Carrying Theory, which is fed with a novel narrowing-based,
fold/unfold program transformation methodology. The core trans-
formation rules are folding, unfolding, definition introduction, def-
nition elimination, and abstraction. The correctness of the program
transformation framework guarantees that the transformed program
is equivalent to the initial one in the sense of Theorem 3.1 (Section
3.3). As already mentioned, this framework opens up new appli-
cations to program optimization, program synthesis, program spe-
cialization and theorem proving for first-order typed rule-based lan-
guages such as ASF+SDF, Elan, OBJ, CafeOBJ, and Maude.

We have shown that our methodology can be effectively applied
to CCT and may significantly simplify the code producer task.
Actually, by applying a strategy such as composition, tupling, etc., the
code producer can (semi-)automatically obtain the final improved
program. Of course, for the methodology to pay off in practice,
the transformation system could be instrumented to also provide
an estimation of the achieved optimization. However, this is out
of the scope of this paper. We intend to investigate such an exten-
sion in a future work. Thanks to the fact that our transformation
methodology relies on narrowing, this can be done by adapting
the narrowing-based transformation strategies of [1]. Moreover,
as an outcome of the transformation process, a compact representa-
tion of the sequence of applied transformation rules is delivered
as a certificate to the code consumer. The code consumer needs
only applying the certificate to the initial requirements to obtain
the desired program. This checking can be completely automated.
We provided a prototypical implementation of the transformation
framework written in Maude. Some available Maude formal tools
can be employed to verify relevant program properties required by
our methodology such as termination [19] of the equation set, con-
We are now implementing the complete CCT infrastructure along
with several fully automatic transformation strategies on top of the
transformation system. Moreover, we are developing a number of
transformation templates in the sense of [10] for rewriting logic
theories, whose correctness should (hopefully) follow from our re-

The prototype is freely available along with some examples at
http://users.dimi.uniud.it/~michele.baggi/cct/
Figure 2. Snapshot of the transformation system interface written in Maude.


A. Proofs of the Theorems

THEOREM A.1. Let $R$ be a FUN theory and $R'$ a rewrite theory which has been obtained from $R$ by means of the application of one transformation rule selected from introduction, elimination, fold, unfold. Then $R'$ is also a FUN theory.

PROOF A.1. It is easy to verify that all the conditions enforced over rewrite theories are preserved by the transformation rules. Since the set of equational axioms $B$ is not changed by transformation rules, condition over $B$ are always preserved. The conditions over coherence and consistency are assured by the preprocessing of the rewrite theories and the constraints over the sets of defined symbols as explained in Appendix B. Right linearity of equations and rules is preserved since the eurekas have to be linear and the fold operation inserts in the right hand side a linear instance of the eureka left hand side. The sort decreasing property is also preserved by fold (by conditions (3) and (4) over the fold operation) and unfold (by the narrowing correctness). The sufficient completeness is preserved by unfold thanks to narrowing completeness.

DEFINITION A.1. Given a FUN transformation sequence of the form $(R_0, ..., R_k)$, we define a virtual FUN transformation sequence $(R'_0, ..., R'_n)$ as a FUN transformation sequence satisfying the following:

(1) $R'_0 = R_0 \cup R_{\text{new}}$, where $R_{\text{new}}$ contains all the eureka equations and rules introduced in $(R_0, ..., R_k)$.

(2) The sequence $(R'_0, ..., R'_n)$ is constructed by applying only the rules: unfolding, folding, and abstraction, in the same order as in the original transformation sequence.

(3) If some definition has been eliminated in $(R_0, ..., R_k)$, then by simply eliminating the same definitions in $R'_n$ we obtain exactly $R_k$.

In practice, only folding and unfolding rules are considered, since abstraction is recast in terms of definition introduction and folding.

LEMMMA A.1. Let $E, E'$ be two equational theories satisfying properties (i) - (iv), (viii) and (ix), such that $E'$ is obtained from $E$ by a fold operation over an equation $E$. Then $\text{GFF}_E = B \iff \text{GFF}_{E'}$.

PROOF A.2. Let $E$ be an equation of the form $(t = r)$ while the eureka $E_c$, used to fold equation $E$, be of the form $(t_0 = r_0)$. From the definition of the fold operation it follows that $t_0 = r_0 \sigma_c$ for some position $p \in \mathbb{N}V\mathbb{P}\text{os}(r)$ and the result of the folding operation of $E$ by $E_c$ is the equation $E': (l = r[t_0, \sigma_c])$. Finally, $\Delta' = \Delta = \{E\} \cup \{E'\}$.

We want to prove that, given any ground term $t$, if $t \not\rightarrow^*_B s$ then $t \not\rightarrow^*_B s'$, and $s = \beta s''$. We will prove it by induction on the length of the rewriting sequence in $\Delta$.

(n = 0) This case is immediate since $t = B s$.

(n > 0) Let us decompose the rewriting sequence form $t$ to $s$ as follows: $t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$. On the rewriting sequence from $t_1$ to $s$ we can apply the induction hypothesis and we can concentrate on the first rewriting step. If $t$ rewrites to $t_1$ without using equation $E$, the same step can be performed in $\Delta'$ and the claim holds.

Otherwise, if equation $E$ is used for the first step, it means that (i) $t_0 = B \sigma_1$, and (ii) $t_1 = [r_1, \sigma_1]_{(p)}$. Then, considering term $t$ and equation $E$, we note that, by (i), it is possible in $E'$ a rewriting step from $t$ using $E$, thus obtaining term $t_2 = [r_1, \sigma_1]_{(p')}$.

We can apply the induction hypothesis and we now concentrate on the first rewriting step. If $t_1$ rewrites to $t_2$, we can show how the considered rewrite step using $E$ can be simulated in $\Delta'$ and the claim holds.

We want to prove that, given any ground term $t$, if $t \not\rightarrow^*_B s$ then $t \not\rightarrow^*_A s$, and $s = \beta s''$. We will prove it by induction on the length of the rewriting sequence in $\Delta$.

(n = 0) This case is immediate since $t = \beta s''$.

(n > 0) Let us decompose the rewriting sequence form $t$ to $s'$ as follows: $t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$.

If $t$ rewrites to $t_1$ without using equation $E$, the same step can be performed in $\Delta$ and, by applying the induction hypothesis on the rewriting sequence from $t_1$ to $s'$, the claim holds. Otherwise, if equation $E$ is used for the first step, then term $t_1$ will embed an instance of the left-hand side ($l_c$) of equation $E_c$. If the following rewrite step from $t_1$ in $\Delta'$ uses the equation $E_c$, we can show how the considered rewrite step $t \rightarrow E$, $t_1 \rightarrow E_c$, $t_2$ can be simulated in $\Delta$ by only one rewrite step from $t$ using equation $E$. While, if the rewrite step form $t_1$ does not use equation $E_c$, since $E'$ is confluent, we can consider a different rewrite sequence from $t$ to $s'$ where $E_c$ is used to rewrite $t_1$, and then use the induction on the length of this new rewrite sequence.

So, let us consider that $E_c$ is used to rewrite $t_1$ in $t_2$ in the rewrite sequence to $s'$. Then, if $t_1 = [r_1, \sigma_1]_{(p)}$, $t_2$ will be the term $[r_1, \sigma_1]_{(p')}$. Applying equation $E$ to term $t_2$ we will obtain term $t_3 = [r_1, \sigma_1]_{(p')}$. Since $t_0 = B \sigma_c$ we have that $t_3 = B [r_1, \sigma_1]_{(p')} = t_2$. The application of the induction hypothesis on the rewrite sequence from $t_2$ to $s'$ completes the proof.

LEMMA A.2. Let $E, E'$ be two equational theories satisfying properties (i) - (iv), (viii) and (ix), such that $E'$ is obtained from $E$ by an unfold operation over an equation $E$. Then $\text{GFF}_E = B \iff \text{GFF}_{E'}$.

PROOF A.3. Let $E$ be an equation of the form $(l = r)$ and $E_0, \ldots, E_k$ the sets of equations used to unfold $E$, each one of the form $(l_i = r_i)$ for $i = 0, \ldots, k$. Let $f_1, \ldots, f_n$ with $n \leq k$ be the set of symbols defined by equations $E_1, \ldots, E_k$. Also let $r \rightarrow_\alpha \Delta, B r_j$ ($j \in \{1, \ldots, n\}$) be the $\Delta, B$-narrowing step such that the
result of unfolding \( E \) using \( E_j \) is the equation \( E_j^{\tau} : (l_\sigma = r_j^\tau) \). From the definition and the correctness of narrowing, we recall that:

1. \( \forall \tau, \sigma_j \to E_j^\tau r_j^\tau \)
2. \( \forall \) there exists position \( p_j \in \mathcal{NPV}(r) \) such that \( r_j^{\tau}[p_j, \sigma_j] = B \)
3. \( \forall \tau, r_j^\tau = (r[p_j])[\sigma_j] \)

\[ \begin{align*}
\text{We want to prove that, given any ground term } t, \text{ if } t \to \Delta_b s \text{ then } t \to \Delta, B s', \text{ and } s = B s'. \text{ From } t \to \Delta_b s \text{ and the con-} \\
\text{fluence and termination of } E, \text{ there exists a rewrite sequence} \quad \text{from } t \to s \text{ where at each step the left-most inner-most redex is} \\
\text{reduced. We will prove the result by induction on the length of such a rewrite sequence.} \\
\end{align*} \]

\[ \begin{align*}
(n \geq 1) \text{ This case is immediate since } t \equiv B s. \\
(n \geq 0) \text{ Let us decompose the rewriting sequence form } t \to s \text{ as follows: } t \to t_i \to s. \text{ On the rewrite sequence from } t_i \to s \text{ to } s \text{ we can apply the induction hypothesis and we now concentrate} \\
\text{on the first rewriting step. If } t \text{ rewrites to } t_i \text{ without using equation } E, \text{ the same step can be performed in } R' \text{ and} \\
\text{the claim holds. Otherwise, if rule } R \text{ is used for the first step, it means that } \text{(i) } t \psi = \sigma_0, \text{ and (ii) } t \equiv E t_i \psi. \text{ Then,} \\
\text{considering term } t \text{ and rule } R_i, \text{ we note that, by (i), it is possible in } R' \text{ a rewrite step from } t \text{ using } R_i, \text{ thus obtaining} \\
\text{a term } t \equiv E t[\sigma_i] \psi. \text{ Propagating substitution } \sigma_i \text{ we} \\
\text{obtain } t \equiv E t[\sigma_i][l_\sigma[\sigma_i]] \psi. \text{ Since term } l_i \text{ is included} \\
\text{in } t \equiv E t_i \psi, \text{ a rewriting step using } R_i \text{ is also possible and we obtain} \\
t \equiv E t[\sigma_i][l_\sigma[\sigma_i]] \psi. \text{ Since } t \psi = E r_\sigma, \text{ we have that} \\
t \equiv E t[\sigma_i][l_\sigma[\sigma_i]] \psi = E t_1 \psi = E t_1, \text{ which completes the proof.} \\
\end{align*} \]

\[ \begin{align*}
\text{We want to prove that, given any ground term } t, \text{ if } t \to \Delta' s' \text{ then } t \to \Delta s', \text{ and } s = E s'. \text{ We will prove it by induction on the length of the rewriting sequence in } R'. \\
\end{align*} \]

\[ \begin{align*}
\text{(n \geq 0) This case is immediate since } t \equiv E s'. \\
\text{(n \geq 0) Let us decompose the rewriting sequence form } t \to s \text{ as follows: } t \to t_i \to s. \text{ On the rewrite sequence from } t_i \to s \text{ we can apply the induction hypothesis and we now concentrate} \\
\text{on the first rewriting step. If } t \text{ rewrites to } t_i \text{ without using rule } R_j, \text{ the same step can be performed in } R \text{ and the claim holds.} \\
\text{Otherwise, if rule } R_j \text{ is used for the first step, term } t_1 \text{ will embed the rewrite } l_i \sigma_j. \text{ Note that for the required properties on the rules defining an eureka, } R_e \text{ is the only rule applicable to} \\
\text{reduce the considered redex. Considering the rewrite sequence from } t_i \to s, \text{ any rule application can} \\
\text{a. reduce the redex } l_j \sigma_j, \text{ if the rule is } R_j; \\
\text{b. reduce a redex in } \sigma_j; \\
\text{c. reduce a redex that does not contain } l_i \sigma_j; \\
\text{d. reduce a redex that contains } l_i \sigma_j \text{ erasing it from the term} \\
\text{(i.e. the variable matching the subterm containing } l_i \sigma_j \text{ does not occur in the rhs of the rule);} \\
\text{e. reduce a redex that contains } l_i \sigma_j \text{ without erasing it.} \\
\end{align*} \]

Recalling that } t_1 \text{ is rewritten up to normal form, either case a. or d. has to occur first until } s' \text{ is reached. In the former case} \\
\text{we can anticipate the application of rule } R_e \text{ to } t_1 \text{ to reduce the redex and show that a subsequent application of rules } R_j \text{ and } R_e \text{ can be simulated in } R \text{ with one application of} \\
\text{rule } R. \text{ In the latter, since the eureka symbol is erased, note that we can just replace the application of rule } R_j \text{ with } R \text{ in } \\
\text{obtaining } t_1 \equiv E t[\sigma_i] \psi \text{ and delete from the sequence all rules that reduce a redex in } \sigma_j \text{ which are} \\
\text{useless. In both cases we reduce the rewrite sequence at least one step, so we can apply the induction hypothesis on the} \\
\text{rest of the sequence. Now we can show how the rewrite steps} \\
t \to R_i \Delta, B t_2 \text{ can be simulated in } R \text{ by only one rewrite step from } t \text{ using rule } R. \\
\text{If } t_1 \equiv E t[\sigma_i] \psi, \text{ and } t_2 \equiv E t[\sigma_i][l_\sigma[\sigma_i]] \psi, \text{ by} \\
\text{applying rule } R \text{ to term } t \text{ we obtain term } t_3 \equiv E t[\sigma_i][l_\sigma[\sigma_i]] \psi. \text{ Since} \\
\text{and we have that } t_3 \equiv E t[\sigma_i][l_\sigma[\sigma_i]] \psi = E t_2. \text{ The}
application of the induction hypothesis on the rewrite sequence from \( t_2 \) to \( s \) completes the proof.

**Lemma A.4.** Let \( R_i, R'_i \) be two FUN theories such that \( R'_i \) is obtained from \( R_i \) by an unfolding operation over a rule \( R \). Then \( \text{GNF}_{R_i} = \text{GNF}_{R'_i} \).

**Proof.** Let \( R_i \) be a rule of the form \( (i \rightarrow r) \) and \( R_1, \ldots, R_k \) the sets of rules used to unfold rule \( R_i \), each one of the form \( (i \rightarrow r_i) \) for \( i = 1, \ldots, k \). Let \( f_1, \ldots, f_n \) with \( n \leq k \) the set of symbols defined by rules \( R_1, \ldots, R_k \). Also let \( \tau \sim_{\sigma_j} R_i \Delta_B \) for \( j \in \{1, \ldots, n\} \), be the \( \Delta \) B-narrowing step such that the result of unfolding \( R_i \) using \( R_j \) is the rule \( R_j' := \{ (\sigma_j \rightarrow r_j) \} \). From the definition and the correctness of narrowing, we recall that:

1. \( \forall j \cdot \tau \sigma_j \rightarrow R_j \)
2. \( \forall j \cdot \exists \sigma_j \in \text{\textsc{Nvp}} \tau (\text{\textsc{r}}) \) such that \( \tau \sigma_j = B \)
3. \( \forall j \cdot \tau \sigma_j' = (\tau \sigma_j \rho) \)

We want to prove that, given any ground term \( t \), if \( t \rightarrow \_ R_i \) then \( t \rightarrow \_ R_i \) and \( s = \_ s. \) We will prove it by induction on the length of the rewriting sequence in \( R_i \).

**Case (1).** This case is immediate since \( t \rightarrow \_ R_i = \_ s. \) We decompose the rewrite sequence \( t \rightarrow \_ s. \) as follows: \( t \rightarrow t_1 \rightarrow \_ s. \) On the rewrite sequence from \( t_1 \) to \( s \) we can apply the induction hypothesis and we now concentrate on the first rewrite step. If \( t \) rewrites to \( t_1 \) without using rule \( R_i \), the same step can be performed in \( R_i \) and the claim holds.

Otherwise, we want to describe a procedure to reorder an initial fragment of the rewrite sequence \( t \rightarrow \_ s. \) in such a way that it is then trivial to simulate it in \( R_i \) and then use the induction hypothesis on the rest of the sequence. Consider a ground term \( w \) and a subsequent application of rules \( R_i \) and \( R_j \) in \( R_i \) as follows:

\[ w \rightarrow (\text{\textsc{r}} \rightarrow \_ B) \]

Then, we obtain an \( E \)-equivalent term to \( w \tau \theta \), which embeds a ground instance of \( \tau \). So, such term contains some occurrences of the symbols \( f_1, \ldots, f_n \). Then, if we can apply \( (R_j) \cup E \) for some \( j \in \{1, \ldots, k\} \), to reduce the redex having one such symbol as its root, we obtain an \( E \)-equivalent term to \( w \tau \sigma_j \rho \), where \( \sigma_j \) is a substitution that unifies the subterm \( \text{\textsc{r}} \) by replacing \( \theta \) with some \( \beta \) that unifies \( \tau \). Now, the rewrite step \( R_i \rightarrow \_ s. \) is the first component of the pair of lists returned and \( R_j \rightarrow \_ s. \) can be simulated in \( R_i \) by an application of rule \( R_i \). In fact, since the rewrite step using \( R_j \) occurs at position \( p_1 \in \text{\textsc{Nvp}} (\tau) \), it follows that the left-hand side \( l_1 \) of rule \( R_j \) unifies with the subtree \( \{ \text{\textsc{r}} \} \) by substitution \( \sigma_j \), which unifies \( \theta \) and the rewritten step \( \tau \sigma_j \rightarrow \_ \) can be proven in \( R_j \). By the definition of unfolding, the rule \( \sigma_j \rightarrow \_ \) is one \( R_j \) belonging to \( R_i \). Finally, by applying \( (R_j) \cup E \) to \( w \tau \sigma_j \rho \), we obtain an \( \text{\textsc{r}} \)-equivalent term to \( w \tau \sigma_j \rho \).

The basic aim of the sequence reordering procedure \( \text{\textsc{rorderSeq}} \) of Figure A is to change the rule application order such obtaining an equivalent sequence (in the sense that the same normal form \( s \) is reached) where the application of rule \( R_i \) is immediately followed by an application of a rule \( R_j \). In the procedure a rewrite sequence is presented as a list of rewrite steps \( (R, p) \) where \( R \) is the applied rule and \( p \) the position of the reduced redex. Each rewrite step is intended to be followed by a \( \Delta \)-normalization. The procedure takes as input the rewrite sequence starting from the rewrite step using rule \( R_i \) and returns the reordered rewrite sequence. List \( s_1 \) contains the reordered portion of the sequence which can be easily simulated in \( R_i \) while \( s_2 \) contains the rest of the sequence (if any). The auxiliary procedure \( \text{\textsc{rorder}} \) uses two auxiliary lists \( n \) and \( vs \). The former contains the sequence of steps that are moved before \( (R_i, p) \), while the latter contains the skipped steps during the reordering, that will keep the same position in the final rewrite sequence. The final sequence is made up with the \( n \) list, the consecutive steps \( (R_i, p), (R_j, p_j) \), the skipped steps in \( vs \) and the rest of the sequence in \( ts \). There is only one particular case in which the reordering procedure deletes some rewrite steps including the one using rule \( R_i \), which will be discussed later. Let us explain the seven different cases of the ordering procedure in the \( \text{\textsc{rorder}} \) function.

Case (1) is the easiest one because the applied rule is one \( R_j \), used to reduce a redex in \( \tau \_ R_i \) having one symbol \( f_i \) at its root. In such a case the procedure terminates returning the reordered sequence \( n \_ s = \_ s \). Case (2) is different from \( R_i \) as used to reduce a redex in the substitution \( \theta \). Note that, such a rewrite step is possible before the application of rule \( R_i \) and hence it is moved at the end of the \( n \) list and the procedure follows with the rest of the sequence. Case (7) is analogous because a rule different from \( R_i \) is used to reduce a redex that contains the term \( \tau \_ R_i \) without erasing it. Such a rewrite step can also be moved before the application of rule \( R_i \) and hence it is put at the end of the \( n \) list. Case (4) considers a rule that reduces a redex whose root is not in \( \tau \_ R_i \) nor in a path from \( p \) to the term root. This is the case of a skippable rewrite step that is moved at the end of the \( n \) list. Case (5) considers a rewrite step where the reduced redex contains term \( \tau \_ R_i \) but erases it from the term (i.e., the variable matching the term \( \tau \_ R_i \) does not occur in the rhs of the rule). Such a rule application makes all the rewrite steps stored in \( n \) and the one using \( R_i \) useless, so they can be deleted from the sequence and the procedure terminates returning the step \( R_i \), the skipped steps, and the rest of the sequence. Cases (2) and (6) consider a rewrite step where the same rule \( R_i \) is used to reduce a redex inside \( \theta \), or containing the subterm \( \tau \_ R_i \), respectively. The basic idea is that when another application of \( R_i \) is found, we first terminate the reordering w.r.t. the deeper application of \( R_i \) and then we recursively call the \( \text{\textsc{rorder}} \) function to reorder the sequence w.r.t. the less deep \( R_i \) application. In fact, in case (2) we suspend the reordering procedure w.r.t. the considered application of rule \( R_i \) and we recursively call the function to reorder a fragment of the rewrite sequence w.r.t. the deeper \( R_i \) application. When the recursive call terminates we resume the previous call putting at the end of the \( n \) list the computed \( s_1 \) list and following with the computed rest of the sequence \( ts \). Case (6) does the viceversa, by terminating the current reordering and then recursively calling the function w.r.t. the less deep \( R_i \) application.

**Termination.** Since the considered rewrite sequence ends with the normal form \( s \), the occurrences of symbols \( f_1, \ldots, f_n \) have to be reduced before reaching \( s \) by using either a rule \( R_j \) as considered in case (2), or a rule that makes them disappear as considered in case (5). In both cases the \( \text{\textsc{rorder}} \) procedure terminates.

**Correctness.** We want to show that all the rewrite steps contained in list \( s_1 \) (which is then merged with the rest of the sequence \( s_2 \) in function \( \text{\textsc{rorderSeq}} \)), can be trivially simulated in \( R_i \). List \( s_1 \) is the first component of the pair of lists returned by the \( \text{\textsc{rorder}} \) function. Considering the termination cases, the first component can contain only the step \( (R_i, \theta) \) (case (5)) where \( R_i \) is different from \( R_i \), or the list \( \text{\textsc{r}} \_ s = \_ s \). In case (1) note that a rewrite step \( (R_i, \theta) \) contained in \( n \_ s \) is such that \( R_i \neq R_i \), or the following rewrite step uses a rule \( R_j \), see cases (1), (3), (6), and (7). Recalling that a subsequent application of rules \( R_i \) and \( R_i \) can be simulated in \( R_i \) by an application of rule \( R_i \), the correctness holds.

**Reduction of the sequence.** It is easy to see that \( s_1 \) is never empty and the rest of the sequence \( s_2 \) is strictly shorter that the
reorderSeq(R^n, p : seq) = let (s_1, s_2) = reduce((R^n, p), [], [], seq) in merge(s_1, s_2)
reorder((R^n, p), ns, vs, (R, q) : ts)
if q = p_j ∈ NVP(\rho) and R = R_j, then return (\langle ns, (R^n, p), (R_j, p_j), [\langle vs \rangle] \rangle, [\langle vs \rangle, ts])
if q ∉ NVP(\rho) and R = R^n, then let (s_1, s_2) = reorder((R^n, q), [\langle vs \rangle, ts]) in reorder((R^n, p), [\langle ns, vs \rangle], [\langle vs, ts \rangle])
if q ∉ NVP(\rho) and R = R^n, then reorder((R^n, p), [\langle ns, (R, q) \rangle], [\langle vs \rangle, ts])
if q ≠ p and q ∉ p_j, then reorder((R^n, p), ns, [\langle (R, q) \rangle], ts)
if q < p, then reorder((R^n, p), [\langle (R, q) \rangle], [\langle vs \rangle, ts])
if q < p, then reorder((R^n, p), [\langle (R, q) \rangle], [\langle vs \rangle, ts])

Figure 3. Rewrite sequence reordering procedure.

sequence from t_1 to s. Hence we can use the inductive hypothesis on s_2.
Exhaustiveness of the cases. Note that the only case in which the considered rule R applies at a position q ≥ p and q ∈ P(\rho) can be the one where q = p_j for some j ∈ \{1, ..., n\} (case (1)). This is in fact the only possibility due to the evil rule constraint considered in the unfolding operation. In other words, if R applies at a position q ∈ NVP(\rho) and q ≠ p_j for all j, then R is an evil rule for R^n and that means that the unfolding operation would not have been performed.

We want to prove that, given any ground term \alpha, then t_\alpha → \alpha'.
We prove it by induction on the length of the rewriting sequence R^\prime.

Lemma A.6. Let (R_0, ..., R_k), k > 0 be a FUN transformation sequence. Then, for all t ∈ T_{R_0}, if \alpha = t_\alpha \in red_{R_0}(t) then there exist s_1, s_2 such that s_1 ∈ GNF_{R_k}(t), s_2 ∈ GNF_{R_0}(s) and s_1 = R_0 s_2. We want to prove that, given any ground term \alpha, then t_\alpha → \alpha'.
We prove it by induction on the length of the rewriting sequence R^\prime.

Theorem A.2. Let (R_0, ..., R_k), k > 0, be a virtual FUN transformation sequence. Then GNF_{E_0} = B GNF_{E_1}, and GNF_{R_0} = E_0 GNF_{R_k}.

Proof A.6. The proof follows immediately from Lemma A.1 and A.2, and Lemma A.3 and A.4, since, at each of the first i-th transformation steps (0 ≤ i ≤ k) a fold or unfold operation has been performed over the equations, and at each of the following k - i transformation steps, a fold or unfold operation has been performed over rules.

Lemma A.5. Let R, R' be two FUN theories such that R' is obtained from R by a fold or unfold operation over a rule R. Then, for all t ∈ T_{R'}, if \alpha = t_\alpha \in red_{R'}(t) then there exist s_1, s_2 such that s_1 ∈ GNF_{R_0}(t), s_2 ∈ GNF_{R'_0}(s) and s_1 = R_0 s_2. Viceversa, for all t ∈ T_{R'}, if \alpha = t_\alpha \in red_{R'}(t) then there exist s_1, s_2 such that s_1 ∈ GNF_{R_0}(t), s_2 ∈ GNF_{R'_0}(s) and s_1 = R_0 s_2.

Proof A.7(\Rightarrow) We will prove it by induction on the length of the rewrite sequence from t to s in R.
(n = 0). Since R' is weakly normalizing, there exists s_1 ∈ GNF_{R_0}(t). Since from Lemma A.3 and A.4 GNF_{R_0} = E GNF_{R'_0}, there exists s_2 ∈ GNF_{R_0}(t) such that s_1 = R_0 s_2 and the claim holds.
(n > 0). Let us decompose the rewriting sequence form t to s as follows: t → t_1 → ... t_n → s. On the rewriting sequence from t_1 to s we can apply the induction hypothesis thus obtaining terms s_1, s_2 such that s_1 ∈ GNF_{R'_0}(t), s_2 ∈ GNF_{R_0}(s) and s_1 = R_0 s_2. Since now we have a rewrite sequence from t to a normal form s_2 and from Lemma A.3 and A.4 GNF_{E_0} = E GNF_{R'_0}, then there exists term s_3 ∈ GNF_{R_1}(t) such that s_2 = E s_3, and the proof is done.

Proof A.8. In order to prove the property we just need to show that we can extend the property of Lemma A.5 to three FUN theories and hence it will hold for a generic k > 0.
Consider the FUN theories R_0, R_1, R_2. By Lemma A.5 we have that for all t ∈ T_{R_0}, if \alpha = t_\alpha \in red_{R_0}(t) then there exist s_1, s_2 such that s_1 ∈ GNF_{R_1}(t), s_2 ∈ GNF_{R_2}(s) and s_1 = R_0 s_2. From Theorem A.2 we have that GNF_{E_0} = B GNF_{E_1} = B GNF_{R_2}, and GNF_{R_0} = E_0 GNF_{R_1} = E_0 GNF_{R_2}. Then, since s_1 ∈ GNF_{R_1}(t), there exists term s_3 ∈ GNF_{R_2}(t) such that s_3 = E_0 s_3. Summing up we have that t ∈ red_{R_0}(t) and there exists s_2, s_3 such that s_3 ∈ GNF_{R_2}(t), s_2 ∈ GNF_{R_0}(s), and s_2 = E_0 s_3, which concludes the proof.

Finally, since each FUN transformation sequence can be transformed into an equivalent virtual FUN transformation sequence (following Definition A.1) which produces the same output program, we have the following Theorem by combining Theorem A.2 and Lemma A.6.

Corollary A.1 (Strong Correctness). Let (R_0, ..., R_k), k > 0 be a FUN transformation sequence. Then, GNF_{E_0} = B GNF_{E_1} = B GNF_{R_2}, GNF_{R_0} = E_0 GNF_{R_1} and for all t ∈ T_{R_0}, if \alpha = t_\alpha \in red_{R_0}(t) then there exist s_1, s_2 such that s_1 ∈ GNF_{R_0}(t), s_2 ∈ GNF_{R_0}(s) and s_1 = R_0 s_2. Viceversa, for all t ∈ T_{R_0}, if \alpha = t_\alpha \in red_{R_0}(t) then there exist s_1, s_2 such that s_1 ∈ GNF_{R_0}(t), s_2 ∈ GNF_{R_0}(s) and s_1 = E_0 s_2.

B. Coherence and Consistence
We propose here a method to guarantee the coherence of →_A,B with B and the E-consistence of →_A,B with B when associativity (A), commutativity (C) and unity (U) axioms (or a subset of them) are considered. The procedure consists of adding some extension
variables to each equation or rule having in the left-hand side a
topmost symbol which is declared with a subset of the \{A, C, U\} axioms, thus obtaining a new set of generalized rules.

For instance, assume we declare the operator + and provide it
with only the associativity axiom. Consider now a rule \( r : x + x \rightarrow 0 \), and a term \( t : a + (a + b) \). Then \( \rightarrow_{r, B} \) is not A-coherent since
there is no matching between term \( t \) and the left-hand side of \( r \),
whereas the A-equivalent term \( t' : (a+a)+b \) matches the left-hand
side by means of substitution \( \{ x/a \} \). To make \( \rightarrow_{r, B} \) A-coherent
we need to add some extension variables \( y \) and \( z \) thus producing
the following set of rules:

\[
\begin{align*}
x + x & \rightarrow 0 \\
x + x + y & \rightarrow 0 + y \\
z + x + x & \rightarrow z + 0 \\
z + x + x + y & \rightarrow z + 0 + y
\end{align*}
\]

Now, given any term \( t \) with topmost symbol +, \( t \) admits a rewriting step with one of these rules iff for each term \( t' \) which is A-
equivalent to \( t \), \( t' \) admits a rewriting step too. If operator + is de-
declared with both A and C axioms, then to make \( \rightarrow_{r, B} \) AC-coherent
we need to introduce only the following two rules:

\[
\begin{align*}
x + x & \rightarrow 0 \\
x + x + y & \rightarrow 0 + y
\end{align*}
\]

Finally, if also the identity axiom (U) is declared for +, then only rule

\[ x + x + y \rightarrow 0 + y \]

is needed for ACU-coherence. Now it is quite easy to derive the
needed generalization for any subset of axioms \{A, C, U\}. A similar transformation can be defined for the case of the equations.

In Maude, this generalization does not have to be performed
explicitly as a transformation of the specification, because it is
achieved implicitly in a built-in, automated way.

For what concern the \( \rightarrow_{R, B} E \)-consistence with \( \rightarrow_{\Delta, B} \) we want
to show how this is guaranteed by the disjointness of the sets of
defined symbols \( D_1 \) and \( D_2 \) (see Sec 2) and the fact that symbols in
\( D_1 \) can not appear in the lhs of rewrite rules. Consider a rewrite
theory and (i) a rewrite step \( t_1 \rightarrow_{R, B} t_2 \) using a rule \( R : l \rightarrow r \)
and (ii) \( t_1 \rightarrow_{\Delta, B} t_3 \). From (i) it follows that there exist substitution \( \theta \) and position \( p \) such that \( t_1 \upharpoonright p = B \_l\theta \) and \( t_2 \equiv_{\Delta, B} t_1 \upharpoonright r\theta \upharpoonright p \). Since
symbols in \( D_1 \) can not appear in \( l \), all possible steps using \( \Delta, B \) from \( t_1 \) can not modify the structure of \( l \) embedded in \( t_1 \), that will then appear unmodified in \( t_3 \). So, after the \( \Delta, B \) steps, there would exists substitution \( \rho \equiv_{\Delta, B} \theta \) such that \( t_3 \upharpoonright p = B \_l\rho \) and hence it is possible a rewrite step \( t_3 \rightarrow_{R, B} t_4 \) such that \( t_4 \equiv_{E} t_2 \).