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

María Alpuente
DSIC, Universidad Politécnica 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 $\mathcal{R}_0, \ldots, \mathcal{R}_n$—where each program $\mathcal{R}_i$ is obtained from the preceding ones $\mathcal{R}_0, \ldots, \mathcal{R}_{i-1}$ 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-

[*] This work has been partially supported by the EU (FEDER) and the Spanish MEC, under grant TIN2007-68093-C02-02, and the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004.

clared to obey given algebraic laws (the so-called *equational attributes* of OBJ, CafeOBJ and Maude), whose effect is to compute with equivalence classes modulo such axioms while avoiding the risk of non–termination. Similarly, theorem provers, both general first-order logic ones and inductive theorem provers, routinely support commonly occurring equational theories for some function symbols such as associativity-commutativity. Moreover, several of the above-mentioned languages and provers have an expressive *order-sorted* typed setting with sorts and subsorts (where subsort inclusions form a partial order and are interpreted semantically as set-theoretic inclusions of the corresponding data sets). Surprisingly, to the best of our knowledge there seems to be no program transformation framework coping with sorts, rules, equational theories, and algebraic laws. In this paper, we develop our 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 other useful elements such as narrowing (a combination of 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 and proofs induced by user specifications. As opposed to more traditional certification approaches such as proof–carrying code [32], where both the code and the certificate are transmitted from the code producer to the code consumer, in CCT only the certificate is transmitted in the form of a theory (a set of axioms and theorems) together with a set of proofs of the theorems; no code needs to be explicitly transmitted. The code consumer would admit new axioms from the code producer only if the associated proofs actually do prove the theorems. If this checking succeeds, then the code consumer can apply a code extractor to the set of function-defining axioms to obtain the executable code. The form of the function-defining axioms is such that it is easy to extract executable code from them. By using the proposed system of fold/unfold transformations, which can be applied to a wide class of programs automatically, our CCT methodology greatly reduces the burden on the code producer.

  The key idea behind our CCT methodology is as follows. Assuming the code consumer provides the requirements in the form of a rewrite theory, the code producer can (semi-) automatically obtain an efficient implementation of the specified functions by applying a sequence of transformation rules. Moreover, having proved the correctness of the transformation system, the code producer can transmit as the required certificate just a compact representation of the sequence of transformation rules to the consumer so he does not need to manually construct any other correctness proof. By applying the transformation rules to the initial requirements, the code consumer can inexpensively obtain the executable code that can be eventually compiled to a different target language if needed.
- We provided an implementation of our transformation methodology, and we conducted an experimental evaluation of the system, which demonstrates the usefulness of our approach.

Related work. In [40, 41], an implementation of CCT has been presented using ATHENA [2, 3], which is a tool that provides a language for both ordinary computation and for logical deduction. To the best of our knowledge, no other implementation of CCT has been proposed in the related literature. The system presented in [41] requires *manually* defining a set of axioms, which are the basis for providing an efficient implementation of the specified functions; then, a proof of the correctness conditions required by the consumer is *manually* constructed using the previously defined axioms.

A different approach to program transformation is proposed in [10], where a term–rewriting transformation framework is formalized by using templates. 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* Σ, with a finite poset of sorts $(S, \leq)$. We assume an S-sorted family $\mathcal{X} = \{\mathcal{X}_s\}_{s \in S}$ of disjoint variable sets. $\mathcal{T}_\Sigma(\mathcal{X})_s$ and $\mathcal{T}_{\Sigma s}$ are the sets of terms and ground terms of sort s, respectively. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding term algebras. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. We write $\overline{o_n}$ 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 (Λ 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 $\mathcal{P}os(t)$ and $\mathcal{NVP}os(t)$ 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 σ is a mapping from variables to terms $\{x_1/t_1, \ldots, x_n/t_n\}$ such that $x_i\sigma = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $x\sigma = x$ for all other variables x.

An *(order-sorted) equational theory* is a pair $E = (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a collection of equations ($l = r$), and B is a collection of equational axioms for associativity, commutativity, and identity declared for the different defined functions. We assume Σ can be always considered as the disjoint union $\Sigma = \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$, called *constructors*, and symbols $f \in \mathcal{D}$, called *defined functions*, each one having a fixed arity, where $\mathcal{D} = \{f \mid f(\bar{t}) = r \in \Delta\}$ and $\mathcal{C} = \Sigma - \mathcal{D}$. Then $\mathcal{T}(\mathcal{C}, \mathcal{X})$ 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 Δ together with the equational axioms of B in an equational theory E induce a congruence relation on the set of terms $\mathcal{T}_\Sigma(\mathcal{X})$ which is usually denoted by $=_E$. E is a presentation or axiomatization of $=_E$. 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 σ is a *E-unifier* of two generic terms t and t' if $t\sigma$ and $t'\sigma$ are both reduced to the same canonical form modulo the equational theory (in symbols $t\sigma =_E t'\sigma$).

A *(order-sorted) rewrite theory* is a triple $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, where Σ is the union $\mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{C}_1 \cup \mathcal{C}_2$ such that $\mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset$, and $(\mathcal{D}_1 \uplus \mathcal{C}_1, \Delta \cup B)$ is an order-sorted equational theory. R is a set of rewrite rules of the form $l \to r$ such that l does not contain any symbol of $\mathcal{D}_1$, $\mathcal{D}_2 = \{f \mid f(\bar{t}) \to r \in R\}$, and $\mathcal{C}_2$ is the set of constructor symbols used in R. Symbols in $\mathcal{D}_2$ are called *defined symbols* as well as those in $\mathcal{D}_1$, 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 $\mathcal{C}_1$ and $\mathcal{C}_2$. We omit Σ when no confusion can arise. Given a rule $l \to r$, 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 $t \to t'$ are said to be:

(1) *Non-erasing*, if $\mathcal{V}ar(t) = \mathcal{V}ar(t')$.

(2) *Sort preserving*, if for each substitution σ, we have $t\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_s$ if and only if $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_s$.

(3) *Sort decreasing*, if for each substitution σ, $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_s$ implies $t\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_s$.

(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 are a position $p_0 \in \mathcal{NVP}os(l_0)$ and substitution σ such that $l_0|_{p_0}\sigma =_B l_1\sigma$, or, viceversa, a position $p_1 \in \mathcal{NVP}os(l_1)$ and substitution σ such that $l_1|_{p_1}\sigma =_B l_0\sigma$. 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 $\mathcal{T}_\Sigma(\mathcal{X})$ as follows: $t \to_R t'$ if there is a position $p \in \mathcal{P}os(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p = l\sigma$ and $t' = t[r\sigma]_p$. An instance $l\sigma$ of a rule $l \to r$ is called a *redex*. A term t without redexes is called *normal form*. A rewrite theory $\mathcal{R}$ is *weakly normalizing* if every term t has a normal form in $\mathcal{R}$, though infinite rewrite sequences from t may exist. The relation $\to_{R/E}$ for rewriting modulo E is defined as $=_E \circ \to_R \circ =_E$. Let $\to \subseteq A \times A$ be a binary relation on a set A. We denote the transitive closure by $\to^+$, the reflexive and transitive closure by $\to^*$, and the rewrite up to normal form by $\to^!$.

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 subsort relation and overloading, sufficient completeness for weakly-normalizing, confluent and sort decreasing rewrite theories, can be checked by showing that for each term $f(t_1, \ldots, t_n)$ where $f : s_1 \ldots s_n \to s$ 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* $\mathcal{C}_1 = \{b, e\}$, $\mathcal{C}_2 = \{b, c, d\}$, $\mathcal{D}_1 = \{a, d\}$, $\mathcal{D}_2 = \{f\}$, $\Delta = \{a = b, d = e\}$, $R = \{f(b, c) \to d\}$ *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* $f(c, a) =_\Delta f(c, b) =_B f(b, c) \to_R d =_\Delta e$.

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 Δ 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 $\to_{R/E}$, since E-congruence classes can be infinite, $\to_{R/E}$-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) Δ is *sort decreasing*, and *confluent and terminating modulo B*.

We define the relation $\to_{\Delta,B}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ as follows: $t \to_{\Delta,B} t'$ if there is a position $p \in \mathcal{P}os(t)$, $l = r$ in Δ, and a substitution σ such that $t|_p =_B l\sigma$ and $t' = t[r\sigma]_p$. The relation $\to_{R,B}$ is similarly defined, and we define $\to_{R\cup\Delta,B}$ as $\to_{R,B} \cup \to_{\Delta,B}$. The idea is to implement $\to_{R/E}$ using $\to_{R\cup\Delta,B}$. For this approach to be correct and complete, we need the following additional assumptions [30].

(iv) $\to_{\Delta,B}$ is *coherent with B*, i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \to^+_{\Delta,B} t_2$ and that $t_1 =_B t_3$ implies $\exists t_4, t_5$ such that $t_2 \to^*_{\Delta,B} t_4$, $t_3 \to^+_{\Delta,B} t_5$, and $t_4 =_E t_5$.

(v) $\to_{R,B}$ is *E-consistent with B*, i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \to_{R,B} t_2$ and that $t_1 =_B t_3$ implies $\exists t_4$ such that $t_3 \to_{R,B} t_4$, and $t_2 =_E t_4$.

(vi) $\rightarrow_{R,B}$ is *E-consistent with* $\rightarrow_{\Delta,B}$, i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \rightarrow_{R,B} t_2$ and that $t_1 \rightarrow^*_{\Delta,B} t_3$ implies $\exists t_4, t_5$ such that $t_3 \rightarrow^*_{\Delta,B} t_4$, $t_4 \rightarrow_{R,B} t_5$, and $t_5 =_E t_2$[1].

Narrowing [21] generalizes term rewriting by allowing free variables in terms (as in logic programming) and by performing unification (at non-variable positions) instead of matching in order to (non–deterministically) reduce a term. The narrowing relation for rewriting logic theories is defined as follows [30].

DEFINITION 3.1 ($R \cup \Delta, B$-Narrowing). *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be an order-sorted rewrite theory satisfying properties (i) - (vi) above. The* $R \cup \Delta, B$-narrowing *relation on* $\mathcal{T}_\Sigma(\mathcal{X})$ *is defined as* $t \rightsquigarrow_{\sigma, R\cup\Delta, B} t'$ *if there exist* $p \in \mathcal{NVP}os(t)$*, a rule* $l \rightarrow r$ *or equation* $l = r$ *in* $R \cup \Delta$*, and* σ*, which is a B-unifier of* $t|_p$ *and* l *such that* $t' = (t[r]_p)\sigma$*.* $t \rightsquigarrow_{\sigma, R\cup\Delta, B} t'$ *is also called a* $R \cup \Delta, B$-narrowing step.

Moreover, to ensure narrowing completeness we assume that (vii) B is *linear* and (viii) R is *sort decreasing* and *right-linear*. Properties (i) - (viii) are enforced on rewrite theories to guarantee the strong reachability-completeness of the narrowing relation $\rightsquigarrow_{\sigma, R\cup\Delta, B}$. Roughly speaking, for each term t and substitution θ such that $t\theta \rightarrow_{\mathcal{R}} t_1$, then narrowing is guaranteed to find another substitution σ such that for some ρ we have $\theta \rightarrow^*_{\mathcal{R}} \rho$, and σ is more general than ρ. In other words, $t_1 \rightarrow^*_{\mathcal{R}} t_2$, $t\sigma \rightarrow_{\mathcal{R}} t_3$, and for some substitution η, $t_3\eta =_E t_2$.

3.2 Transformation rules

Let us define a *FUN* (Fold/Unfold traNsformable) theory 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 $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1}, \ldots, \mathcal{R}_k)$, $k \geq 0$, where each $\mathcal{R}_j$ 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 Δ_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 $f(\overline{t_i}) = r_i$ (*resp.* $f(\overline{t_i}) \rightarrow r_i$), 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 $f : s_1 \ldots s_n \rightarrow s$ `[Ax]`, where $s_1, \ldots, s_n, s$ are sorts declared in $\mathcal{R}_0$ and `Ax` are equational attributes.

(2) $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, and $\mathcal{V}ar(\overline{t_i}) = \mathcal{V}ar(r_i)$, 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 completeness 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 $f(x) \rightarrow g(x)$ using the (non left linear) eureka $new(x, x) \rightarrow g(x)$, which would produce a new rule $f(x) \rightarrow new(x, x)$ 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 $\mathcal{R}_k$,

- all equations that define the functions $f_0, \ldots, f_n$, say Δ^f, such that $f_0, \ldots, f_n$ do not occur either in $\mathcal{R}_0$ or in $(\Sigma_k, (\Delta_k \backslash \Delta^f) \cup B_k, R_k)$.
- all rules that define the functions $f_0, \ldots, f_n$, say R^f, such that $f_0, \ldots, f_n$ do not occur either in $\mathcal{R}_0$ or in $(\Sigma_k, \Delta_k \cup B_k, R_k \setminus R^f)$.

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

$$\{l\sigma = r' \mid r \rightsquigarrow_{\sigma, \Delta_k, B_k} r' \text{ is a } \textit{narrowing} \text{ step in } E_k\}.$$

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 $R = l' \rightarrow r' \in \mathcal{R}_k$ is *evil* for F, if there exist a substitution ρ and position $p \in \mathcal{NVP}os(l')$, $p \neq \Lambda$ such that $r\rho = l'|_p\rho$. 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

$$\{l\sigma \rightarrow r' \mid r \rightsquigarrow_{\sigma, R_k \cup \Delta_k, B_k} r' \text{ is a } \textit{narrowing} \text{ step in } \mathcal{R}_k\}.$$

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.

[1] Properties (iv) and (v) can be achieved by a simple preprocessing of rewrite rules, while property (vi) is guaranteed by the disjointness of the sets of defined symbols, as shown in Appendix B.

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 R and*

$$
\begin{array}{llll}
R: & & R': & \\
g_1(x) \to x & & g_1(x) \to x & \\
h(0) \to 0 & & h(0) \to 0 & \\
h(1) \to 0 & & h(1) \to 0 & \\
h(s(x)) \to 0 & & h(s(x)) \to 0 & \\
h(s(g_1(x))) \to 1 & & h(s(g_1(x))) \to 1 & \\
g_2(x) \to s(g_1(x)) & & g_2(x) \to s(x) &
\end{array}
$$

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

The use of narrowing empowers the unfold operation by implicitly embedding the instantiation rule (the operation of the Burstall and Darlington framework [9] 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 R and*

$$
R = \left\{
\begin{array}{rcl}
f(0) & \to & 0 \\
g(x) & \to & s(f(x)) \\
h(0) & \to & 0 \\
h(s(x)) & \to & s(0)
\end{array}
\right.
\quad
R' = \left\{
\begin{array}{rcl}
f(0) & \to & 0 \\
g(0) & \to & s(0) \\
h(0) & \to & 0 \\
h(s(x)) & \to & s(0)
\end{array}
\right.
$$

We get program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ *from* $\mathcal{R}$ *by applying an unfolding step over the rule defining function g in R, performing the narrowing step* $s(f(x)) \rightsquigarrow_{x=0} s(0)$*. Now, the term* $h(g(s(0)))$ *can be rewritten in 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 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 $F' \in \mathcal{R}_j$, $0 \leq j \leq k$, be an equation (the "folding equation") of the form $(l' = r')$, such that $r|_p =_{B_k} r'\sigma$ for some position $p \in \mathcal{NVP}os(r)$ and substitution σ. Note that, since we transform the equations of an equational theory, we consider here the congruence relation $=_{B_k}$ modulo the equational axioms B_k (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 σ is sort decreasing, i.e, if $x \in \mathcal{X}_s$, then $x\sigma \in \mathcal{T}_{\Sigma}(\mathcal{X})_{s'}$ such that $s' \leq s$.

(4) Let $l' = f(\overline{t_n})$ and $r|_p = e$ and let $f(\overline{t_n})$ 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 $(l = r[l'\sigma]_p)$.

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 $(l \to r)$, and let $F' \in \mathcal{R}_j$, $0 \leq j \leq k$, be a rule (the "folding rule") of the form $(l' \to r')$, such that $r|_p =_{E_k} r'\sigma$ for some position $p \in \mathcal{NVP}os(r)$ and substitution σ, 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 $(l \to r[l'\sigma]_p)$. 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:*

$$
\begin{array}{rcll}
new & \to & f & \textit{(eureka)} \\
g & \to & f & \textit{(non-eureka)}
\end{array}
$$

Without conditions (1) and (2), a folding of the eureka rule would be possible, obtaining the new rule $(new \to g)$*, 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 R and*

$$
\begin{array}{rclcrcl}
R: & & & & R': & & \\
f(a,b) & \to & g(a,b) & & f(a,b) & \to & g(m(a),b) \\
f(x,y) & \to & g(x,y) & & f(x,y) & \to & g(x,y) \\
m(a) & \to & a & & m(a) & \to & a \\
m(a) & \to & b & & m(a) & \to & b \\
m(b) & \to & a & & m(b) & \to & a \\
g(a,x) & \to & a & & g(a,x) & \to & a \\
g(b,x) & \to & b & & g(b,x) & \to & b
\end{array}
$$

We get program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ *from* $\mathcal{R}$ *by applying a fold step to the rule* $f(a,b) \to g(a,b)$ *using the eureka* $m(a) \to a$*. It is easy to see that in* $\mathcal{R}'$ *we can reduce term* $f(a,b)$ *to the normal forms* a *or* b*, while 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* $g(b,b)$ *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

$$\begin{array}{l} double\_sum(x, y) = ds\_aux(sum(x, y)) \\ ds\_aux(z) = sum(z, z) \end{array}$$

These equations are generated from the original one by a definition introduction of the eureka ds_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 $\mathtt{red}_{\mathcal{R}} = \{s \mid \exists t \in \mathcal{T}_\Sigma \wedge t \rightarrow^*_{\mathcal{R}} s\}$, 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 $\mathtt{GNF}_{\mathcal{R}} = \{s \mid \exists t \in \mathcal{T}_\Sigma \wedge t \rightarrow^!_{\mathcal{R}} s\} \subseteq \mathtt{red}_{\mathcal{R}}$ 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,* $\mathtt{GNF}_{\mathcal{E}_0} =_B \mathtt{GNF}_{\mathcal{E}_k}, \mathtt{GNF}_{\mathcal{R}_0} =_{E_0} \mathtt{GNF}_{\mathcal{R}_k}$, *and for all* $t \in \mathcal{T}_{\Sigma_0}$, *if* $s \in \mathtt{red}_{\mathcal{R}_0}(t)$ *then there exist* s_1, s_2 *such that* $s_1 \in \mathtt{GNF}_{\mathcal{R}_k}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}_0}(s)$ *and* $s_1 =_{E_0} s_2$. *Viceversa, for all* $t \in \mathcal{T}_{\Sigma_0}$, *if* $s \in \mathtt{red}_{\mathcal{R}_k}(t)$ *then there exist* s_1, s_2 *such that* $s_1 \in \mathtt{GNF}_{\mathcal{R}_0}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}_k}(s)$ *and* $s_1 =_{E_0} s_2$.

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*

$$R = \left\{ \begin{array}{rcl} f(0) & \rightarrow & 0 \\ f(s(x)) & \rightarrow & s(x) \\ g(x) & \rightarrow & s(f(x)) \end{array} \right. \quad R' = \left\{ \begin{array}{rcl} f(0) & \rightarrow & 0 \\ f(s(x)) & \rightarrow & s(x) \\ g(0) & \rightarrow & s(0) \\ g(s(x)) & \rightarrow & s(s(x)) \end{array} \right.$$

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 `En` (resp. `Rn`), 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 `En` (resp. `Rn`), 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-*

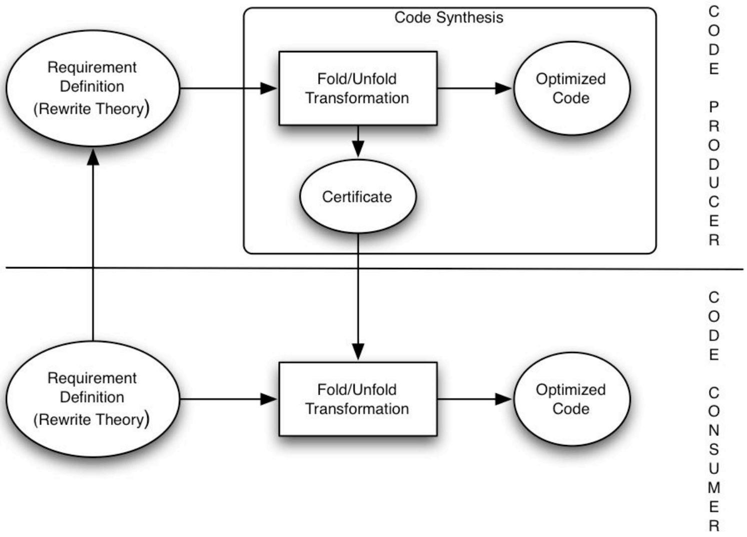

Code Synthesis
Requirement Definition (Rewrite Theory)
Fold/Unfold Transformation
Optimized Code
Certificate
CODE PRODUCER
Requirement Definition (Rewrite Theory)
Fold/Unfold Transformation
Optimized Code
CODE CONSUMER

Figure 1. Code Carrying Theory Architecture Diagram

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* $\mathcal{R}_{i-1}$ *to obtain* $\mathcal{R}_i$.

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 naïve equational theory.*

```
      op fib : Nat -> Nat .
(E1)  eq fib(0) = S(0) .
(E2)  eq fib(S(0)) = S(0) .
(E3)  eq fib(S(S(n))) = fib(S(n)) + fib(n) .
```

Due to the highly recursive nature of this definition of `fib`*, the evaluation of an expression like* `fib(S^50(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* `fib` *with the above rule very inefficient. Let us transform the previous Fibonacci definition into a more efficient one by using the tupling strategy.*
(1) First, we introduce the following eureka which makes use of the `Pair` *data structure*

```
      sort Pair .
      op ⟨_,_⟩ : Nat Nat -> Pair .
      op aux : Nat -> Pair .
(E4)  eq aux(n) = ⟨fib(S(n)),fib(n)⟩ .
```

(2) We now unfold the redex `fib(S(n))` *of equation* `(E4)`.

```
(E5)  eq aux(0) = ⟨S(0),fib(0)⟩ .
(E6)  eq aux(S(n)) = ⟨fib(S(n)) + fib(n),fib(S(n))⟩ .
```

We unfold once again equation `(E5)` *in order to remove the call to* `fib`.

```
(E7)  eq aux(0) =⟨S(0),S(0)⟩ .
```

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

```
      op aux2 : Pair -> Nat .
      op aux3 : Pair -> Pair .
(E8)  eq aux2(⟨x,y⟩) = x+y .
(E9)  eq aux3(⟨x,y⟩) = ⟨x+y,x⟩ .
```

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

```
(E10)  eq fib(S(S(n))) = aux2(⟨fib(S(n)),fib(n)⟩) .
(E11)  eq aux(S(n)) = aux3(⟨fib(S(n)),fib(n)⟩) .
```

(4) Finally, the right-hand sides of both equations are folded using the original definition of function `aux`

```
(E12)  eq fib(S(S(n))) = aux2(aux(n)) .
(E13)  eq aux(S(n)) = aux3(aux(n)) .
```

The transformed (linear) definition of the equational theory for `fib` *is as follows.*

```
(E1)   eq fib(0) = S(0) .
(E2)   eq fib(S(0)) = S(0) .
(E12)  eq fib(S(S(n))) = aux2(aux(n)) .
(E7)   eq aux(0) = ⟨S(0),S(0)⟩ .
(E13)  eq aux(S(n)) = aux3(aux(n)) .
(E8)   eq aux2(⟨x,y⟩) = x+y .
(E9)   eq aux3(⟨x,y⟩) = ⟨x+y,x⟩ .
```

The resulting certificate `C` *is as follows.*

$$\begin{aligned} \mathtt{C} = (&\mathtt{Intro}((\mathrm{op\,aux} : \mathtt{Nat \text{ -> } Pair}.),\\ &(\mathrm{eq\,aux(n)} = \langle \mathtt{fib(S(n)), fib(n)} \rangle.)), \mathtt{Unfold}(\mathtt{E4}, [1]),\\ &\mathtt{Unfold}(\mathtt{E5}, [2]), \mathtt{Intro}((\mathrm{op\,aux2} : \mathtt{Pair \text{ -> } Nat}.),\\ &(\mathrm{eq\,aux2}(\langle \mathrm{x}, \mathrm{y} \rangle) = \mathrm{x} + \mathrm{y}.)),\\ &\mathtt{Intro}((\mathrm{op\,aux3} : \mathtt{Pair \text{ -> } Pair}.),\\ &(\mathrm{eq\,aux3}(\langle \mathrm{x}, \mathrm{y} \rangle) = \langle \mathrm{x} + \mathrm{y}, \mathrm{x} \rangle)), \mathtt{Fold}(\mathtt{E8}, \mathtt{E3}, \Lambda),\\ &\mathtt{Fold}(\mathtt{E9}, \mathtt{E6}, \Lambda), \mathtt{Fold}(\mathtt{E4}, \mathtt{E10}, [1]), \mathtt{Fold}(\mathtt{E4}, \mathtt{E11}, [1])). \end{aligned}$$

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.*

```
      op sum-list : NatList -> Nat .
(R1)  rl sum-list(nil) ⇒ 0 .
(R2)  rl sum-list(x xs) ⇒ fib(x) + sum-list(xs) .
```

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

```
      op sum-list-aux : NatList Nat -> Nat .
(R3)  rl sum-list-aux(xs,x) ⇒ x + sum-list(xs) .
```

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

```
(R4)  rl sum-list-aux(nil,x) ⇒ x .
(R5)  rl sum-list-aux(y ys,x) ⇒ x+fib(y)+sum-list(ys) .
```

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

```
(R1)  rl sum-list(nil) ⇒ 0 .
(R6)  rl sum-list(x xs) ⇒ sum-list-aux(xs,fib(x)) .
(R4)  rl sum-list-aux(nil,x) ⇒ x .
(R7)  rl sum-list-aux(x xs,y) ⇒
                     sum-list-aux(xs,(fib(x) + y)) .
```

The certificate C *is then as follows.*

$$\begin{aligned}\texttt{C} = (&\texttt{Intro}((\texttt{op sum-list-aux : NatList Nat -> Nat.}),\\ &(\texttt{rl sum-list-aux}(\texttt{xs},\texttt{x}) \Rightarrow \texttt{x} + \texttt{sum-list}(\texttt{xs}).)),\\ &\texttt{Unfold}(\texttt{R3},[1.2]),\texttt{Fold}(\texttt{R3},\texttt{R5},\Lambda),\texttt{Fold}(\texttt{R3},\texttt{R2},\Lambda)).\end{aligned}$$

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

5. Implementation

We implemented the transformation framework presented in Section 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 interface is shown in Figure 5. In order to implement the transformation rules, we made use of a useful Maude property called *reflection*. Rewriting logic is reflective in a precise mathematical way. In other words, there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a meta-term $\overline{\mathcal{R}}$, any term t, t' in $\mathcal{R}$ as meta-terms $\overline{t}, \overline{t'}$, and any pair $(\mathcal{R}, t)$ as a meta-term $\langle\overline{\mathcal{R}}, \overline{t}\rangle$, in such a way that we have the following equivalence: $t \rightarrow t'$ in $\mathcal{R}$ iff $\langle\overline{\mathcal{R}}, \overline{t}\rangle \rightarrow \langle\overline{\mathcal{R}}, \overline{t'}\rangle$ in $\mathcal{U}$. Thanks to Maude reflection, our program transformation methodology has been easily implemented by manipulating the meta-term representations of rules and equations. In practice, transformation rules presented in Section 3 have been implemented as rewrite rules that work and manipulate the meta-term representation of the rewrite theories we want to transform. On the other hand, by virtue of our reflective design, our rewrite theory for program transformation is also available to the level of the CCT infrastructure, which allows us to reuse it in a clear and principled way.

Since the unfolding operation uses narrowing, we employed the `META-E-NARROWING` module, which is part of the Full Maude distribution [39]. The narrowing implemented in Maude is called narrowing *with simplification* because it combines the narrowing relation with rewriting to normal form (represented by $\rightarrow^{!}$) . The combined relation ($\rightsquigarrow_{\sigma,R,\Delta\cup B}; \rightarrow^{!}_{\Delta\cup B}$) is defined as $t \rightsquigarrow_{\sigma,R,\Delta\cup B} ; \rightarrow^{!}_{\Delta\cup B} t''$ iff $t \rightsquigarrow_{\sigma,R,\Delta\cup B} t'$, $t' \rightarrow^{*}_{\Delta\cup B} t''$, and t'' is a normal form. For further details, please refer to [13].

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 transformation rules are folding, unfolding, definition introduction, definition 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 applications to program optimization, program synthesis, program specialization and theorem proving for first-order typed rule-based languages 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 extension 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 representation 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, confluence [20] and sufficient completeness [24] of rewrite theories.

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 results easily, while [10] uses an automated theorem prover for the verification.

References

[1] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science*, 311(1-3):479–525, 2004.

[2] K. Arkoudas. *Denotational Proof Languages*. PhD thesis, Massachusetts Institute of Technology, 2000.

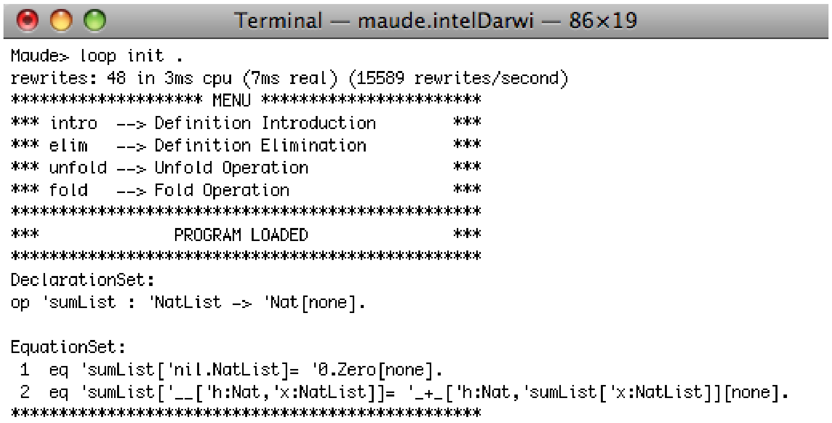
[3] K. Arkoudas. An Athena tutorial, 2005. Available at: `http://www.cag.csail.mit.edu/~kostas/dpls/athena/athenaTutorial.pdf`.

[4] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*. ACM Press, 1989.

[5] P. Borovanský, C. Kirchner, H. Kirchner, and P. E. Moreau. ELAN from a Rewriting Logic Point of View. *Theoretical Computer Science*, 285:155–185, 2002.

[6] A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87, 1993.

[7] A. Bouhoula, J.P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.



```
Maude> loop init .
rewrites: 48 in 3ms cpu (7ms real) (15589 rewrites/second)
******************* MENU ***********************
*** intro  --> Definition Introduction       ***
*** elim   --> Definition Elimination        ***
*** unfold --> Unfold Operation              ***
*** fold   --> Fold Operation                ***
************************************************
***              PROGRAM LOADED              ***
************************************************
DeclarationSet:
op 'sumList : 'NatList -> 'Nat[none].

EquationSet:
 1  eq 'sumList['nil.NatList]= '0.Zero[none].
 2  eq 'sumList['__['h:Nat,'x:NatList]]= '_+_['h:Nat,'sumList['x:NatList]][none].
************************************************
```

Figure 2. Snapshot of the transformation system interface written in Maude.

[8] R. M. Burstall and J. Darlington. Some Transformations for Developing Recursive Programs. *SIGPLAN Not.*, 10(6):465–472, 1975.

[9] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of ACM*, 24(1):44–67, 1977.

[10] Y. Chiba, T. Aoto, and Y. Toyama. Program Transformation by Templates Based on Term Rewriting. In *Proc. of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, (PPDP '05)*, pages 59–69, New York, NY, USA, 2005. ACM.

[11] W. Chin. Towards an Automated Tupling Strategy. In *Proc. of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, (PEPM '93)*, pages 119–132. ACM, 1993.

[12] W. Chin, A. Goh, and S. Khoo. Effective Optimisation of Multiple Traversals in Lazy Languages. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA (Technical Report BRICS-NS-99-1)*, pages 119–130. University of Aarhus, DK, 1999.

[13] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Unification and Narrowing in Maude 2.4. In *Proc. of the 20th International Conference on Rewriting Techniques and Applications, (RTA '09), Brasília, Brazil, 2009*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer-Verlag, 2009.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA '03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 2003.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[16] J. Darlington. *A Semantic Approach to Automatic Program Improvement*. PhD thesis, Department of Machine Intelligence, Edimburgh University, Edimburgh, U.K., 1972.

[17] J. Darlington. Program Transformation. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.

[18] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, AMAST Series, 1998.

[19] F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude Termination Tool (System Description). In *Proc. of the 4th International Joint Conference on Automated Reasoning, (IJCAR '08)*, pages 313–319, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] F. Durán and J. Meseguer. A Church-Rosser Checker Tool for Maude Equational Specifications. Technical report, Universidad de Málaga and SRI International, July 2000.

[21] M. Fay. First Order Unification in an Equational Theory. In *Proc. of 4th International Conference on Automated Deduction*, pages 161–167, 1979.

[22] P. A. Gardner and J. C. Shepherdson. Unfold/Fold Transformation of Logic Programs. In J.L Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 565–583. MIT, 1991.

[23] 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.

[24] J. Hendrix, J. Meseguer, and H. Ohsaki. A Sufficient Completeness Checker for Linear Order-Sorted Specifications Modulo Axioms. In U. Furbach and N. Shankar, editors, *3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer, 2006.

[25] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation. *Theoretical Computer Science*, 75:139–156, 1990.

[26] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[27] M.J. Maher. A Transformation System for Deductive Database Modules with Perfect Model Semantics. *Theoretical Computer Science*, 110(2):377–403, 1993.

[28] Z. Manna and R. J. Waldinger. Toward Automatic Program Synthesis. *Communication of the ACM*, 14(3):151–165, 1971.

[29] N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[30] J. Meseguer and P. Thati. Symbolic Reachability Analysis Using Narrowing and its Application to Verification of Cryptographic Protocols. *Higher Order Symbolic Computation*, 20(1-2):123–160, 2007.

[31] José Meseguer. Conditioned Rewriting Logic as a United Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[32] G.C. Necula. Proof-Carrying Code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL '97)*, pages 106–119, New York, NY, USA, 1997. ACM.

[33] G. Sittampalam O. de Moore. Generic Program Transformation. In *Advanced Functional Programming*, pages 116–149, 1998.

[34] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[35] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[36] D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[37] W. Scherlis. Program Improvement by Internal Specialization. In *Proc. of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL '81)*, pages 41–49, New York, NY, USA, 1981. ACM.

[38] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. of the 2nd International Conference on Logic Programming, (ICLP '84)*, pages 127–139, 1984.

[39] The Maude Team. Full Maude, 2009. Available at: `http://www.lcc.uma.es/~duran/FullMaude/`.

[40] A. Vargun. *Code-Carrying Theory*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 2006.

[41] A. Vargun and D.R. Musser. Code-Carrying Theory. In *ACM Symposium on Applied Computing*, pages 376–383, New York, NY, USA, 2008. ACM.

[42] P. Viry. Rewriting: An effective Model of Concurrency. In *Proc. of the 6th International Conference on Parallel Architectures and Languages Europe, (PARLE '94)*, pages 648–660, London, UK, 1994. Springer-Verlag.

[43] E. Visser. A Survey of Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.

A. Proofs of the Theorems

THEOREM A.1. *Let $\mathcal{R}$ be a FUN theory and $\mathcal{R}'$ a rewrite theory which has been obtained from $\mathcal{R}$ by means of the application of one transformation rule selected from introduction, elimination, fold, unfold. Then $\mathcal{R}'$ 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 consistence 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 $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, we define a* virtual *FUN transformation sequence $(\mathcal{R}'_0, \ldots, \mathcal{R}'_n)$ as a FUN transformation sequence satisfying the following:*

(1) $\mathcal{R}'_0 = \mathcal{R}_0 \cup \mathcal{R}_{new}$, where R_{new} contains all the eureka equations and rules introduced in $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$.

(2) The sequence $(\mathcal{R}'_0, \ldots, \mathcal{R}'_n)$ is constructed by applying only the rules: unfolding, folding, and abstraction[2]*, in the same order as in the original transformation sequence.*

(3) If some definition has been eliminated in $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, then by simply eliminating the same definitions in $\mathcal{R}'_n$ we obtain exactly $\mathcal{R}_k$.

LEMMA A.1. *Let $\mathcal{E}, \mathcal{E}'$ be two equational theories satisfying properties (i) - (iv), (viii) and (ix), such that $\mathcal{E}'$ is obtained from $\mathcal{E}$ by a* fold *operation over an equation E. Then $\mathtt{GNF}_{\mathcal{E}} =_B \mathtt{GNF}_{\mathcal{E}'}$.*

PROOF A.2. *Let E be an equation of the form $(l = r)$ while the eureka E_e, used to fold equation E, be of the form $(l_e = r_e)$. From the definition of the fold operation it follows that $r|_p =_B r_e\sigma$ for some position $p \in \mathcal{NVP}os(r)$ and the result of the folding operation of E by E_e is the equation E_f: $(l = r[l_e\sigma]_p)$. Finally, $\Delta' = \Delta - \{E\} \cup \{E_f\}$.*

$\subseteq$ *We want to prove that, given any ground term t, if $t \to^{!}_{\Delta,B} s$ then $t \to^{!}_{\Delta',B} s'$, and $s =_B s'$. We will prove it by induction on the length of the rewriting sequence in Δ.*
(n = 0.) This case is immediate since $t =_B s$.
($n > 0$.) Let us decompose the rewriting sequence form t to s as follows: $t \to t_1 \to^{!} s$. On the rewriting sequence from t_1 to s we can apply the induction hypothesis and we now concentrate on the first rewriting step. If t rewrites to t_1 without using equation E, the same step can be performed in Δ' and the claim holds. Otherwise, if equation E is used for the first step, it means that (i) $t|_{p'} =_B l\sigma_1$, and (ii) $t_1 = t[r\sigma_1]_{p'}$. Then, considering term t and equation E_f, we note that, by (i), it is possible in $\mathcal{E}'$ a rewriting step from t using E_f, thus obtaining term $t_2 = t[r[l_e\sigma]_p\sigma_1]_{p'}$. Propagating substitution σ_1 we obtain $t_2 = t[r\sigma_1[l_e\sigma\sigma_1]_p]_{p'}$. Since term l_e is embedded in t_2, a rewriting step using E_e is also possible and we obtain $t_3 = t[r\sigma_1[r_e\sigma\sigma_1]_p]_{p'}$. Since $r|_p =_B r_e\sigma$, we have that $t_3 =_B t[r\sigma_1[r|_p\sigma_1]_p]_{p'} = t[r\sigma_1]_{p'} = t_1$, which completes the proof.

$\supseteq$ *We want to prove that, given any ground term t, if $t \to^{!}_{\Delta',B} s'$ then $t \to^{!}_{\Delta,B} s$, and $s =_B s'$. We will prove it by induction on the length of the rewriting sequence in Δ'.*
(n = 0.) This case is immediate since $t =_B s'$.
($n > 0$.) Let us decompose the rewriting sequence form t to s' as follows: $t \to t_1 \to^{!} s'$. If t rewrites to t_1 without using equation E_f, the same step can be performed in Δ and, by applying the induction hypothesis on the rewriting sequence from t_1 to s', the claim holds. Otherwise, if equation E_f is used for the first step, note that term t_1 will embed an instance of the left-hand side (l_e) of equation E_e. If the following rewrite step from t_1 in Δ' uses the equation E_e, we can show how the considered rewrite steps $t \to_{E_f} t_1 \to_{E_e} t_2$ can be simulated in Δ by only one rewrite step from t using equation E. While, if the rewrite step form t_1 does not use equation E_e, since $\mathcal{E}'$ is confluent, we can consider a different rewrite sequence from t to s' where E_e 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_e is used to rewrite t_1 in t_2 in the rewire sequence to s'. Then, if $t_1 = t[r[l_e\sigma]_p\sigma_1]_{p'}$, t_2 will be the term $t[r[r_e\sigma]_p\sigma_1]_{p'}$. Applying equation E to term t we will obtain term $t_3 = t[r\sigma_1]_{p'}$. Since $r|_p =_B r_e\sigma$ we have that $t_3 =_B t[r[r_e\sigma]_p\sigma_1]_{p'} = t_2$. The application of the induction hypothesis on the rewrite sequence from t_2 to s' completes the proof.

LEMMA A.2. *Let $\mathcal{E}, \mathcal{E}'$ be two equational theories satisfying properties (i) - (iv), (viii) and (ix), such that $\mathcal{E}'$ is obtained from $\mathcal{E}$ by a* unfold *operation over an equation E. Then $\mathtt{GNF}_{\mathcal{E}} =_B \mathtt{GNF}_{\mathcal{E}'}$.*

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 $n \leq k$ the set of symbols defined by equations $E_1, \ldots, E_k$. Also let $r \leadsto_{\sigma_j,\Delta,B} r'_j$ $(j \in \{1, \ldots, n\})$ be the Δ, B-narrowing step such that the*

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

result of unfolding E using E_j is the equation $E_j^u : (l\sigma_j = r'_j)$. From the definition and the correctness of narrowing, we recall that:

(1) $\forall j \,.\, r\sigma_j \rightarrow^{E_j} r'_j$
(2) $\forall j$ *there exists position* $p_j \in \mathcal{NVP}os(r)$ *such that* $r|_{p_j}\sigma_j =_B l_j\sigma_j$
(3) $\forall j \,.\, r'_j = (r[r_j]_{p_j})\sigma_j$

$\subseteq$ *We want to prove that, given any ground term t, if $t \rightarrow^!_{\Delta,B} s$ then $t \rightarrow^!_{\Delta',B} s'$, and $s =_B s'$. From $t \rightarrow^!_{\Delta,B} s$ and the confluence and termination of $\mathcal{E}$, there exists a rewrite sequence from t to s where at each step the left-most inner-most redex is reduced. We will prove the result by induction on the length of such a rewrite sequence.*

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

(n > 0.) Let us decompose the rewriting sequence form t to s as follows: $t \rightarrow t_1 \rightarrow^! s$. On the rewriting sequence from t_1 to s we can apply the induction hypothesis and we now concentrate on the first rewriting step. If t rewrites to t_1 without using equation E, the same step can be performed in Δ' and the claim holds. Otherwise, there exists a position $p \in \mathcal{NVP}os(t)$ and a substitution θ such that (i) $l\theta =_B t|_p$, (ii) $t|_p$ is the left-most inner-most redex, and (iii) $t_1 = t[r\theta]_p$. Note that from (ii) and the sufficient completeness of Δ, it follows that (iv) θ is a constructor substitution, that is, for each $x/t \in \theta$, t is a constructor term. From (ii) and (iv) it follows that if $r\theta$ contains a redex, it is the left-most inner-most redex in t_1 and its position p' belongs to $\mathcal{NVP}os(r)$. Since r contains at least one occurrence of the symbols $f_1, \ldots, f_n$, and Δ is sufficient complete, $r\theta$ contains at least one redex. Let p' be the position of the left-most inner-most redex inside t_1. Now, consider the following rewrite step $r\theta \rightarrow_{p',E_j} r[r_j]_{p'}\theta$, $j \in \{1, \ldots, k\}$, which rewrites the redex in position p'. The obtained term t_2 is $t[r[r_j]_{p'}\theta]_p$. Then, since during the unfold operation, we perform narrowing at each possible position in r, the narrowing step $r \rightsquigarrow_{p',\sigma_j,E_j \cup B} r'_j$ can be proven in Δ. By (iv) and the completeness of narrowing, the substitution computed by narrowing is more general then θ, which amounts to say that there exists a substitution ρ such that (v) $\theta = \sigma_j\rho$. By the definition of unfolding, the equation $l\sigma_j = r'_j$ is one E_j^u belonging to Δ'. Finally, from (i) and (v) we can apply the equation E_j^u to term t thus obtaining $t[r'_j\rho]_p = t[((r[r_j]_{p'})\sigma_j)\rho]_p = t[r[r_j]_{p'}\theta]_p = t_2$, and the claim follows by applying the inductive hypothesis to the rewrite sequence from t_2 to s.

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

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

(n > 0.) Let us decompose the rewriting sequence form t to s' as follows: $t \rightarrow t_1 \rightarrow^! s'$. On the rewriting sequence from t_1 to s we can apply the induction hypothesis and we now concentrate on the first rewriting step. If t rewrites to t_1 without using one of the equations E_j^u, the same step can be performed in Δ and the claim holds. Otherwise, if one of the equations E_j^u is used for the last rewriting step, there exists a substitution θ such that $(l\sigma_j)\theta =_B t|_p$, and $t_1 = t[r'_j\theta]_p$. By $r\sigma_j \rightarrow_{E_j} r'_j$ and the stability of rewriting, we have that $(r\sigma_j)\theta \rightarrow_{E_j} r'_j\theta$. Therefore, $t =_B t[l(\sigma_j\theta)]_p \rightarrow_E t[r(\sigma_j\theta)]_p = t[(r\sigma_j)\theta]_p \rightarrow_{E_j} t[r'_j\theta]_p = t_1$, which is a rewrite sequence leading to t_1 in Δ.

LEMMA A.3. *Let $\mathcal{R}, \mathcal{R}'$ be two FUN theories such that $\mathcal{R}'$ is obtained from $\mathcal{R}$ by a* fold *operation over a rule R. Then $\texttt{GNF}_{\mathcal{R}} =_E \texttt{GNF}_{\mathcal{R}'}$.*

PROOF A.4. *Let R be a rule of the form $(l \rightarrow r)$ while the eureka R_e, used to fold rule R, be of the form $(l_e \rightarrow r_e)$. From the definition of the fold operation it follows that $r|_p =_E r_e\sigma$ for some position $p \in \mathcal{NVP}os(r)$ and the result of the folding operation of R by R_e is the rule R_f: $(l \rightarrow r[l_e\sigma]_p)$. Finally, $\mathcal{R}' = \mathcal{R} - \{R\} \cup \{R_f\}$.*

$\subseteq$ *We want to prove that, given any ground term t, if $t \rightarrow^!_{\mathcal{R}} s$ then $t \rightarrow^!_{\mathcal{R}'} s'$, and $s =_E s'$. We will prove it by induction on the length of the rewriting sequence in $\mathcal{R}$.*

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

(n > 0.) Let us decompose the rewriting sequence form t to s as follows: $t \rightarrow t_1 \rightarrow^! s$. On the rewriting sequence from t_1 to s we can apply the induction hypothesis and we now concentrate on the first rewriting step. If t rewrites to t_1 without using rule R, the same step can be performed in $\mathcal{R}'$ and the claim holds. Otherwise, if rule R is used for the first step, it means that (i) $t|_{p'} =_B l\sigma_1$, and (ii) $t_1 =_E t[r\sigma_1]_{p'}$. Then, considering term t and rule R_f, we note that, by (i), it is possible in $\mathcal{R}'$ a rewriting step from t using R_f, thus obtaining a term $t_2 =_E t[r[l_e\sigma]_p\sigma_1]_{p'}$. Propagating substitution σ_1 we obtain $t_2 =_E t[r\sigma_1[l_e\sigma\sigma_1]_p]_{p'}$. Since term l_e is embedded in t_2, a rewriting step using R_e is also possible and we obtain $t_3 =_E t[r\sigma_1[r_e\sigma\sigma1]_p]_{p'}$. Since $r|_p =_E r_e\sigma$, we have that $t_3 =_E t[r\sigma_1[r|_p\sigma1]_p]_{p'} = t[r\sigma_1]_{p'} =_E t_1$, which completes the proof.

$\supseteq$ *We want to prove that, given any ground term t, if $t \rightarrow^!_{\mathcal{R}'} s'$ then $t \rightarrow^!_{\mathcal{R}} s$, and $s =_E s'$. We will prove it by induction on the length of the rewriting sequence in $\mathcal{R}'$.*

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

(n > 0.) Let us decompose the rewriting sequence form t to s' as follows: $t \rightarrow t_1 \rightarrow^! s'$. On the rewriting sequence from t_1 to s we can apply the induction hypothesis and we now concentrate on the first rewriting step. If t rewrites to t_1 without using rule R_f, the same step can be performed in $\mathcal{R}$ and the claim holds. Otherwise, if rule R_f is used for the first step, term t_1 will embed the redex $l_e\sigma_1$. Note that for the required properties on the rules defining an eureka, R_e is the only rule applicable to reduce the considered redex. Considering the rewrite sequence from t_1 to s', any rule application can:

a. *reduce the redex σ_1, if the rule is R_e;*
b. *reduce a redex in σ_1;*
c. *reduce a redex that does not contain $l_e\sigma_1$;*
d. *reduce a redex that contains $l_e\sigma_1$ erasing it from the term (i.e. the variable matching the subterm containing $l_e\sigma_1$ does not occur in the rhs of the rule);*
e. *reduce a redex that contains $l_e\sigma_1$ without erasing it.*

Recalling that t_1 is rewritten up to normal form, either case a. or d. has to occur first until s' is reached. In the former case we can anticipate the application of rule R_e to t_1 to reduce the redex and show that a subsequent application of rules R_f and R_e can be simulated in $\mathcal{R}$ with one application of rule R. In the latter, since the eureka symbol is erased, note that we can just replace the application of rule R_f with R in $\mathcal{R}$ obtaining $t_1 =_E t[r]_{p'}$ and delete from the sequence all the rule applications that reduce a redex in σ_1 which are useless. In both cases we reduce the rewrite sequence of at least one step, so we can apply the induction hypothesis on the rest of the sequence. Now we can show how the rewrite steps $t \rightarrow_{R_f \cup \Delta,B} t_1 \rightarrow_{R_e \cup \Delta,B} t_2$ can be simulated in $\mathcal{R}$ by only one rewrite step from t using rule R.

If $t_1 =_E t[r[l_e\sigma]_p\sigma_1]_{p'}$, and $t_2 =_E t[r[r_e\sigma]_p\sigma_1]_{p'}$, by applying rule R to term t we obtain term $t_3 =_E t[r\sigma_1]_{p'}$. Since $r|_p =_E r_e\sigma$ we have that $t_3 =_E t[r[r_e\sigma]_p\sigma_1]_{p'} =_E t_2$. The application of the induction hypothesis on the rewrite sequence from t_2 to s' completes the proof.

LEMMA A.4. Let $\mathcal{R}, \mathcal{R}'$ be two FUN theories such that $\mathcal{R}'$ is obtained from $\mathcal{R}$ by an unfold operation over a rule R. Then $\mathtt{GNF}_{\mathcal{R}} =_E \mathtt{GNF}_{\mathcal{R}'}$.

PROOF A.5. Let R^u be a rule of the form $(l \to r)$, and $R_1, \ldots, R_k$ the sets of rules used to unfold rule R^u, each one of the form $(l_i \to r_i)$ 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 $r \rightsquigarrow_{\sigma_j, R\cup\Delta, B} r'_j$, $j \in \{1, \ldots, n\}$, be the $R \cup \Delta, B$-narrowing step such that the result of unfolding R^u using R_j is the rule $R^u_j : (l\sigma_j \to r'_j)$. From the definition and the correctness of narrowing, we recall that:

(1) $\forall\, j \,.\, r\sigma_j \to^{R_j} r'_j$
(2) $\forall\, j$ there exists position $p_j \in \mathcal{NVP}os(r)$ such that $r|_{p_j}\sigma_j =_B l_j\sigma_j$
(3) $\forall\, j \,.\, r'_j = (r[r_j]_{p_j})\sigma_j$

$\subseteq$ We want to prove that, given any ground term t, if $t \to^!_{\mathcal{R}} s$ then $t \to^!_{\mathcal{R}'} s'$, and $s =_E s'$. We will prove it by induction on the length the rewriting sequence in $\mathcal{R}$.

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

($n > 0$.) Let us decompose the rewrite sequence form t to s as follows: $t \to t_1 \to^! s$. 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^u, the same step can be performed in $\mathcal{R}'$ and the claim holds. Otherwise, we want to describe a procedure to reorder an initial fragment of the rewrite sequence from t to s in such a way it is then trivial to simulate it in $\mathcal{R}'$ and then use the induction hypothesis on the rest of the sequence. Consider a ground term w and a subsequent application of rules R^u and R_j in $\mathcal{R}$ as follows. If $w|_p =_B l\theta$, by applying $(\{R^u\} \cup E)$ we obtain an E-equivalent term to $w[r\theta]_p$ which embeds a ground instance of r. 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 symbols as its root, we obtain an E-equivalent term to $w[r[r_j]_{p_j}\theta]_p$. The key point that we need is to note that such a subsequent application of rules R^u and R_j in $\mathcal{R}$, can be simulated in $\mathcal{R}'$ by an application of rule R^u_j. In fact, since the rewrite step using R_j occurs at position $p_j \in \mathcal{NVP}os(r)$, it follows that the left hand side l_j of rule R_j unifies with the subterm $r|_{p_j}$ by substitution σ_j that subsumes θ, and therefore the narrowing step $r \rightsquigarrow_{\sigma_j, R\cup\Delta, B} r[r_j]_{p_j}$ can be proven in $R \cup E$. By the definition of unfolding, the rule $l\sigma_j \to r'_j$ is one R^u_j belonging to $\mathcal{R}'$. Finally, by applying (R^u_j, Δ) to term w we obtain an E-equivalent term to $w[r'_j\rho]_p = w[((r[r_j]_{p_j})\sigma_j)\rho]_p = w[r[r_j]_{p_j}\theta]_p$.

The basic aim of the sequence reordering procedure $reorderSeq$ of Figure A is to change the rule application order thus obtaining an equivalent sequence (in the sense that the same normal form s is reached) where the application of rule R^u is immediately followed by an application of a rule R_j. In the procedure a rewrite sequence is represented 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 Δ, B normalization. The procedure takes as input the rewrite sequence starting from the rewrite step using rule R^u and returns the reordered rewrite sequence. List s_1 contains the reordered portion of the sequence which can be easily simulated in $\mathcal{R}'$ while s_2 contains the rest of the sequence (if any). The auxiliary procedure $reorder$ uses two auxiliary lists ns and vs. The former contains the sequence of steps that are moved before (R^u, 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 ns list, the consecutive steps (R^u, 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^u, which will be discussed later. Let us explain the seven different cases of the ordering procedure in the $reorder$ function.

Case (1) is the easiest one because the applied rule is one R_j, used to reduce a redex in $r\theta$ having one symbol f_i at its root. In such a case the procedure terminates returning the reordered sequence $ns, (R^u, p), (R_j, p_j), vs, ts$. In case (3) a rule different from R^u is used to reduce a redex in the substitution θ. Note that, such a rewrite step is possible before the application of rule R^u and hence it is moved at the end of the ns list and the procedure follows with the rest of the sequence. Case (7) is analogous because a rule different from R^u is used to reduce a redex that contains the subterm $r\theta$ without erasing it. Such a rewrite step can also be moved before the application of rule R^u and hence it is put at the end of the ns list. Case (4) considers a rule that reduces a redex whose root is not in $r\theta$ 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 vs list. Case (5) considers a rewrite step where the reduced redex contains term $r\theta$ but erases it from the term (i.e. the variable matching the subterm containing $r\theta$ does not occur in the rhs of the rule). Such a rule application makes all the rewrite steps stored in ns and the one using R^u useless, so they can be deleted from the sequence and the procedure terminates returning the step (R, q), the skipped steps, and the rest of the sequence. Cases (2) and (6) consider a rewrite step where the same rule R^u is used to reduce a redex inside θ, or containing the subterm $r\theta$, respectively. The basic idea is that when another application of R^u is found, we first terminate the reordering w.r.t. the deeper application of R^u and then we recursively call the $reorder$ function to reorder the sequence w.r.t. the less deep R^u application. In fact, in case (2) we suspend the reordering procedure w.r.t. the considered application of rule R^u and we recursively call the function to reorder a fragment of the rewrite sequence w.r.t. the deeper R^u application. When the recursive call terminates we resume the previous call putting at the end of the ns list the computed ns_1 list and following with the computed rest of the sequence ts_1.Case (6) does the viceversa, by terminating the current reordering and then recursively calling the function w.r.t. the less deep R^u 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 $reorder$ 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 $reorderSeq$), can be trivially simulated in $\mathcal{R}'$. List s_1 is the first component of the pair of lists returned by the $reorder$ function. Considering the termination cases, the first component can contain only the step (R, q) (case (5)) where R is different from R^u, or the list $ns, (R^u, p), (R_j, p_j)$ (case (1)). Note that a rewrite step (R, q) contained in list ns is such that $R \neq R^u$, 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^u and R_j can be simulated in $\mathcal{R}'$ by an application of rule R^u_j, 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

```
$reorderSeq((R^u,p) : seq) =$ let $(s_1,s_2) = reduce((R^u,p),[\,],[\,],seq)$ in $merge(s_1,s_2)$
$reorder((R^u,p), ns, vs, (R,q) : ts)$
  if $q = p_j \in \mathcal{NVP}os(r)$ and $R = R_j$, then return $([ns,(R^u,p),(R_j,p_j)],[vs,ts])$   (1)
  if $q \notin \mathcal{NVP}os(r) > p$ and $R = R^u$, then let $(ns_1,ts_1) = reorder((R^u,q),[\,],[\,],ts)$ in $reorder((R^u,p),[ns,ns_1],vs,ts_1)$   (2)
  if $q \notin \mathcal{NVP}os(r) > p$, then $reorder((R^u,p),[ns,(R,q)],vs,ts)$   (3)
  if $q \nleq p$ and $q \ngtr p$, then $reorder((R^u,p),ns,[vs,(R,q)],ts)$   (4)
  if $q < p$ and $f_1,\ldots,f_n$ do not appear in the resulting term, then return $([(R,q)],[vs,ts])$   (5)
  if $q < p$ and $R = R^u$, then let $(ns_1,ts_1) = reorder((R^u,p),ns,vs,ts)$ in $reorder((R^u,q),[ns,ns_1],vs,ts_1)$   (6)
  if $q < p$, then $reorder((R^u,p),[ns,(R,q)],vs,ts)$   (7)
```

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 $q \geq p$ and $q \in \mathcal{P}os(r)$ can be the one where $q = p_j$ for some $j \in \{1,\ldots,n\}$ (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 $q \in \mathcal{NVP}os(r)$ and $q \neq p_j$ for all j, then R is a evil rule for R^u and that means that the unfolding operation would not have been performed.*

$\supseteq$ *We want to prove that, given any ground term t, if $t \rightarrow^!_{\mathcal{R}'} s'$ then $t \rightarrow^!_{\mathcal{R}} s$, and $s =_E s'$. We will prove it by induction on the length of the rewriting sequence in $\mathcal{R}'$.*

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

(n > 0.) Let us decompose the rewriting sequence form t to s' as follows: $t \rightarrow t_1 \rightarrow^! s'$. On the rewriting sequence from t_1 to s we can apply the induction hypothesis and we now concentrate on the first rewriting step. If t rewrites to t_1 without using one of the rules R^u_j, the same step can be performed in $\mathcal{R}$ and the claim holds. Otherwise, if one of the rules R^u_j is used for the last rewriting step, there exists a substitution θ such that $(l\sigma_j)\theta =_B t|_p$, and $t_1 =_E t[r'_j\theta]_p$. By $r\sigma_j \rightarrow^{R_j} r'_j$ and the stability of rewriting, we have that $(r\sigma_j)\theta \rightarrow^{R_j} r'_j\theta$. Therefore, $t =_B t[l(\sigma_j\theta)]_p \rightarrow^{R} t[r(\sigma_j\theta)]_p = t_1[(r\sigma_j)\theta]_p \rightarrow^{R_j} t[r'_j\theta]_p =_E t_1$, which is a rewriting sequence leading to t_1 in $\mathcal{R}$.

THEOREM A.2. *Let $(\mathcal{R}_0,\ldots,\mathcal{R}_k)$, $k > 0$, be a virtual FUN transformation sequence. Then $\mathtt{GNF}_{\mathcal{E}_0} =_B \mathtt{GNF}_{\mathcal{E}_k}$, and $\mathtt{GNF}_{\mathcal{R}_0} =_{E_0} \mathtt{GNF}_{\mathcal{R}_k}$.*

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 ($0 \leq i \leq k$) 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 $\mathcal{R},\mathcal{R}'$ be two FUN theories such that $\mathcal{R}'$ is obtained from $\mathcal{R}$ by a* fold *or* unfold *operation over a rule R. Then, for all $t \in \mathcal{T}_\Sigma$, if $s \in \mathtt{red}_{\mathcal{R}}(t)$ then there exist s_1, s_2 such that $s_1 \in \mathtt{GNF}_{\mathcal{R}'}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}}(s)$ and $s_1 =_E s_2$. Viceversa, for all $t \in \mathcal{T}_\Sigma$, if $s \in \mathtt{red}_{\mathcal{R}'}(t)$ then there exist s_1, s_2 such that $s_1 \in \mathtt{GNF}_{\mathcal{R}}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}'}(s)$ and $s_1 =_E s_2$.*

PROOF A.7*($\Rightarrow$) We will prove it by induction on the length of the rewrite sequence from t to s in $\mathcal{R}$.*

(n=0.) Since $\mathcal{R}'$ is weakly normalizing, there exists $s_1 \in \mathtt{GNF}_{\mathcal{R}'}(t)$. Since from Lemma A.3 and A.4 $\mathtt{GNF}_{\mathcal{R}} =_E \mathtt{GNF}_{\mathcal{R}'}$, there exists $s_2 \in \mathtt{GNF}_{\mathcal{R}}(t)$ such that $s_1 =_E s_2$ and the claim holds.

(n > 0.) Let us decompose the rewriting sequence form t to s as follows: $t \rightarrow t_1 \rightarrow^! s$. On the rewriting sequence from t_1 to s we can apply the induction hypothesis thus obtaining terms s_1, s_2 such that $s_1 \in \mathtt{GNF}_{\mathcal{R}'}(t_1)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}}(s)$ and $s_1 =_E s_2$. Since now we have a rewrite sequence from t to a normal form s_2 and from Lemma A.3 and A.4 $\mathtt{GNF}_{\mathcal{R}} =_E \mathtt{GNF}_{\mathcal{R}'}$, then there exists term $s_3 \in \mathtt{GNF}_{\mathcal{R}'}(t)$ such that $s_2 =_E s_3$, and the proof is done.

($\Leftarrow$) It is analogous to the previous direction.

LEMMA A.6. *Let $(\mathcal{R}_0,\ldots,\mathcal{R}_k)$, $k > 0$ be a FUN transformation sequence. Then, for all $t \in \mathcal{T}_{\Sigma_0}$, if $s \in \mathtt{red}_{\mathcal{R}_0}(t)$ then there exist s_1, s_2 such that $s_1 \in \mathtt{GNF}_{\mathcal{R}_k}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}_0}(s)$ and $s_1 =_{E_0} s_2$. Viceversa, for all $t \in \mathcal{T}_{\Sigma_0}$, if $s \in \mathtt{red}_{\mathcal{R}_k}(t)$ then there exist s_1, s_2 such that $s_1 \in \mathtt{GNF}_{\mathcal{R}_0}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}_k}(s)$ and $s_1 =_{E_0} s_2$.*

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 $\mathcal{R}_0,\mathcal{R}_1,\mathcal{R}_2$. By Lemma A.5 we have that for all $t \in \mathcal{T}_\Sigma$, if $s \in \mathtt{red}_{\mathcal{R}_0}(t)$ then there exist s_1, s_2 such that $s_1 \in \mathtt{GNF}_{\mathcal{R}_1}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}_0}(s)$, and $s_1 =_{E_0} s_2$. From Theorem A.2 we have that $\mathtt{GNF}_{\mathcal{E}_0} =_B \mathtt{GNF}_{\mathcal{E}_1} =_B \mathtt{GNF}_{\mathcal{E}_2}$ and $\mathtt{GNF}_{\mathcal{R}_0} =_{E_0} \mathtt{GNF}_{\mathcal{R}_1} =_{E_0} \mathtt{GNF}_{\mathcal{R}_2}$. Then, since $s_1 \in \mathtt{GNF}_{\mathcal{R}_1}(t)$, there exists term $s_3 \in \mathtt{GNF}_{\mathcal{R}_2}(t)$ such that $s_2 =_{E_0} s_3$. Summing up we have that $t \in \mathtt{red}_{\mathcal{R}_0}(t)$ and there exists s_2, s_3 such that $s_3 \in \mathtt{GNF}_{\mathcal{R}_2}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}_0}(s)$, 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 $(\mathcal{R}_0,\ldots,\mathcal{R}_k)$, $k > 0$ be a FUN transformation sequence. Then, $\mathtt{GNF}_{\mathcal{E}_0} =_B \mathtt{GNF}_{\mathcal{E}_k}, \mathtt{GNF}_{\mathcal{R}_0} =_{E_0} \mathtt{GNF}_{\mathcal{R}_k}$, and for all $t \in \mathcal{T}_{\Sigma_0}$, if $s \in \mathtt{red}_{\mathcal{R}_0}(t)$ then there exist s_1, s_2 such that $s_1 \in \mathtt{GNF}_{\mathcal{R}_k}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}_0}(s)$ and $s_1 =_{E_0} s_2$. Viceversa, for all $t \in \mathcal{T}_{\Sigma_0}$, if $s \in \mathtt{red}_{\mathcal{R}_k}(t)$ then there exist s_1, s_2 such that $s_1 \in \mathtt{GNF}_{\mathcal{R}_0}(t)$, $s_2 \in \mathtt{GNF}_{\mathcal{R}_k}(s)$ and $s_1 =_{E_0} s_2$.*

B. Coherence and Consistence

We propose here a method to guarantee the coherence of $\rightarrow_{\Delta,B}$ with B and the E-consistence of $\rightarrow_{R,B}$ 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 $r : x+x \to 0$, and a term $t : a + (a + b)$. Then $\to_{r,B}$ 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 $\to_{r,B}$ A-coherent we need to add some *extension variables* y and z thus producing the following set of rules:

$$\begin{gathered} x + x \to 0 \\ x + x + y \to 0 + y \\ z + x + x \to z + 0 \\ z + x + x + y \to z + 0 + y \end{gathered}$$

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 declared with both A and C axioms, then to make $\to_{r,B}$ AC-coherent we need to introduce only the following two rules:

$$\begin{gathered} x + x \to 0 \\ x + x + y \to 0 + y. \end{gathered}$$

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

$$x + x + y \to 0 + y$$

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 $\to_{R,B}$ E-consistence with $\to_{\Delta,B}$ we want to show how this is guaranteed by the disjointness of the sets of defined symbols $\mathcal{D}_1$ and $\mathcal{D}_2$ (see Sec 2) and the fact that symbols in $\mathcal{D}_1$ can not appear in the lhs of rewrite rules. Consider a rewrite theory and (i) a rewrite step $t_1 \to_{R,B} t_2$ using a rule $R : l \to r$ and (ii) $t_1 \to^*_{\Delta,B} t_3$. From (i) it follows that there exist substitution θ and position p such that $t_1|_p =_B l\theta$ and $t_2 =_{\Delta,B} t_1[r\theta]_p$. Since symbols in $\mathcal{D}_1$ can nor appear in l, all possible steps using Δ, 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 Δ, B steps, there would exists substitution $\rho =_{\Delta,B} \theta$ such that $t_3|_p =_B l\rho$ and hence it is possible a rewrite step $t_3 \to_{R,B} t_4$ such that $t_4 =_E t_2$.