Program Transformation and Rewriting

Francoise Bellegarde

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-021

September, 1990

Program Transformation and Rewriting

Françoise BELLEGARDE
Oregon Graduate Institute of Science and Technology
19600 NW von Neumann Dr.
Beaverton OR. 97006-1999
email: bellegar@cse.ogi.edu

Abstract

We present a basis for program transformation using term rewriting tools. A specification is expressed hierarchically by successive enrichments as a signature and a set of equations. A term can be computed by rewriting. Transformations come from applying a partial unfailing completion procedure to the original set of equations augmented by inductive theorems and a definition of a new function symbol following diverse heuristics. Moreover, the system must provide tools to prove inductive properties; to verify that enrichment produces neither junk nor confusion; and to check for ground confluence and termination. We show how these properties are related to the correctness of the transformation.

1 Equations and program transformation

An important research topic in the area of automatic programming is transformational programming. Functional programming is not inhibited by superfluous concerns such as sequential control or storage mapping. Transformational programming offers a means to formally develop efficient programs from clear programs expressed in high level, functional languages. The program transformation paradigm is not new, but it can take its place in software design only if the transformation process is automated as much as possible. Another condition for program transformation to become a useful method for software design is that it can be used to transform large programs.

A problem with the transformation paradigm is the loss of visibility of its design. Hand transformation is a lengthy, boring and error-prone process. Transformation systems might help by making the process semi-automatic, but this is not enough. As the form of a program is changed during the transformation process, its meaning soon becomes unclear and the user gets lost.

On the other hand, the program transformation process requires knowledge about the program. Properties of the program direct its transformation, and the programmer must provide them. Sometimes, these properties are well known, and they do not need to be proved over and over again. Sometimes, the proof is easy to do by hand. This task may or may not take place during the transformation process itself. In any case, the transformation system must be able to take account of properties given by the user. Moreover, it must be able to help to prove or disprove some of the properties suggested by the programmer.

1.1 Equations in functional programs

Either a purely functional fragment of a language like ML or a fragment of an order-sorted language like OBJ [10] can be considered as a good candidate for a specification language. It is relatively easy to write or to translate a specification with such languages in an equational form. We will consider a *specification* given by:

- a signature Σ composed of a set of *sort symbols* and a set of *function symbols* with rank declarations.
- a set E of equations.

In this sense, a specification describes a class of algebras, namely the class of Σ-algebras satisfying the equations E. But the semantics we give to such specifications is the *initial algebra* $\Im(\Sigma, E)$.

Example 1 The following specification describes *append*:

$$
\begin{aligned}
&sort\ list[elem] \\
&nil : \mapsto list \\
&:: : elem \times list \mapsto list \\
&append\ :\ list \times list \mapsto list \\
&\forall x : list.append(nil, x)\ =\ x \\
&\forall x : elem.xs, y : list.append((x :: xs), y)\ =\ x :: append(xs, y)
\end{aligned}
$$

The possibility of describing and transforming an application by successive *enrichments* of a specification allows us to handle large programs.

Definition 1 *An enrichment of a specification $S = (\Sigma, E)$ is a specification $S' = (\Sigma', E')$ such that $\Sigma \subseteq \Sigma'$ and $E \subseteq E'$.*

Enrichments can produce either junk, that is new terms that are not equivalent to an already existing term, or confusion, that is equivalence between two terms originally distinct.

Example 2 We can enrich the specification of the example 1 by adding the inductive equations:

$$\forall x, y, z : list.append(x, append(y, z)){=}append(append(x, y), z) \tag{1}$$

$$\forall x : list.append(x, nil){=}x \tag{2}$$

We can also enrich the specification of Example 1 by adding a new function *reverse* and equations for its definition:

$$reverse : list \mapsto list$$

$$reverse(nil)\ =\ nil \tag{3}$$

$$\forall x : elem.xs : list.reverse(x :: xs)\ =\ append(reverse(xs), x :: nil) \tag{4}$$

These two enrichments do not create junk or confusion.

For now, we consider only the particular case of a pure sorted equational language. Some extensions could be considered in the future, such as equations conditioned by premises. Our goal is to define what a transformation system based on rewriting tools can offer. Before going further, let us give basic notions and notations that are used in this paper.

1.2 Basic notions and notations

We will denote by $T(\Sigma, X)$ the set of terms built with the variables X and the functions symbols of the signature Σ. The set of ground terms or terms without variables is denoted by $T(\Sigma)$. Positions in terms are represented as a sequence of integers. t/p denotes the subterm of t at the position p. Substitutions are endomorphisms of $T(\Sigma, X)$. The replacement of the subterm t/p in t by the term u is denoted by $t[p \leftarrow u]$.

Given a binary relation, $\rightarrow$, $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$. $\leftrightarrow^*$ is its reflexive and symmetric transitive closure. A relation $\rightarrow$ is noetherian if there is no infinite sequence $t_1 \rightarrow t_2 \cdots$. A relation $\rightarrow$ is confluent if $\leftarrow^* \circ \rightarrow^* \subseteq \rightarrow^* \circ \leftarrow^*$, where $\circ$ denotes the composition of relations. An equation is a pair of terms $s = t$. Given a set E of equations, we write $s \leftrightarrow_E t$ if $s/p = \sigma(l)$ and $t = s[p < -\sigma(r)]$ for some position p in t, substitution σ and equation $l = r$ or $r = l$ in E.

A rule is an oriented pair of terms $l \rightarrow r$. A term rewriting system is a set of rules. Given a term rewriting system R, the rewriting relation $\rightarrow_R$ is a binary relation in $T(\Sigma, X)$. $s \rightarrow_R t$ if there exists a rule $l \rightarrow r$ in R, a position p in s, a substitution σ such that $\sigma(l) = s/p$ and $t = s[p < -\sigma(r)]$. A term t is in normal form if it is irreducible.

A term rewriting system is terminating if the relation $\rightarrow_R$ is noetherian, confluent if the relation $\rightarrow_R$ is confluent, and convergent if it is both confluent and terminating. Convergence ensures existence and unicity of the normal form of every term.

Critical pairs are produced by overlaps of two redexes in a same term. A non-variable term t' and a term t overlap if there exists a non-variable position p in t such that t/p and t' are unifiable. Let $g \rightarrow d$ and $l \rightarrow r$ be two rules such that l and g overlap at the position p with the most general unifier σ. The overlapped term $\sigma(g)$ produces the critical pair (p, q) defined by $p = \sigma(g[p < -r])$ and $q = \sigma(d)$. A critical pair is convergent if p and q reduce to the same term.

The *completion procedure* [12] was introduced as a means at deriving convergent term-rewriting systems used as procedures for deciding the validity of identities (the word problem) in a given equational theory. The procedure generates new rewrite rules to resolve ambiguities resulting from existing rules that overlap. These new rules are produced by non-convergent critical pairs.

A completion procedure can fail because it is unable to orient an equation into a rule without losing the termination property of the system. However, non-orientable equations may sometimes be used for reduction anyway, because their instances can be oriented. This idea is basic to the unfailing completion procedure [2, 1]. It uses the notion of ordered rewriting which does not require that an equation always be used from left to right. An ordered rewriting system is a set of equations together with a reduction ordering $>$, i.e. a well-founded, monotonic and stable. An ordered rewriting system can be denoted $(E, >)$. When the equations in E can be oriented with $>$, we usually call them rules. The ordered rewriting relation using $(E, >)$ is the rewriting relation $\rightarrow_{E>}$ where $E>$ denotes the set of all the orientable instances of E. This allows us to extend the notion of critical pairs to ordered critical pairs and to extend the completion process to an unfailing completion process, i.e. a completion that cannot fail. The outcome of the unfailing completion procedure, when it does not loop, is either a (ground) convergent term rewriting system R when all equations are rules or a ground convergent ordered rewriting system $(E, >)$ when some equations remain unordered. By *ground convergence*, we mean termination and confluence on ground terms. Obviously, convergence implies ground convergence.

Given a ground convergent term rewriting system R, a term t is *ground (or inductively)* *reducible* with R if all its ground instances are R reducible.

An equation $s = t$ is an *inductive theorem (or inductive consequence)* of E if for any ground substitution σ, $\sigma(s) = \sigma(t)$.

1.3 Checking properties of enrichments

Using equational logic as a programming language was proposed by O'Donnell [17], by Gogen [10] and by Dershowitz [8]. An operational semantics can be given to functions defined by equations by using term rewriting systems.

We consider programs presented in a specification $S = (\Sigma, E)$ by a set of equations E. The specification S is constructed by successive enrichments of a specification $S_0 = (\Sigma_0, E_0)$. We consider the case when the set of functions in the signature Σ can be split into a set of constructors C and a set of defined functions D. The definition of functions of D is *sufficiently complete with respect to* C, i.e. it produces no junk, if every ground term is provably equal to a *constructor term*, which is a term built only with constructors.

When E can be partitioned into constructors and defined symbols, $E_C \cup E_D$, where E_C is the subset of equations that contain only constructors and variables. If $E_C = \emptyset$, the constructors are said to be free. The specification is consistent with respect to C, i.e. it produces no confusions, if for all constructor terms s and t, $s \longleftrightarrow_E^* t$ iff $s \longleftrightarrow_{E_C}^* t$. A good transformation system must be able to prove properties about specifications. Let us consider the principal results regarding enrichments.

Let $S = (\Sigma, E) \subseteq S' = (\Sigma, E')$ be an enrichment with only new equations: $E' = E \cup E_0$. The enrichment is consistent if every equation in E_0 is an inductive consequence of E.

When theories are presented by ground convergent term rewriting systems, the ground completion process can be used to prove consistency of an enrichment and to produce simultaneously a ground convergent term rewriting system for the enriched specification. Consider an enrichment $S = (\Sigma, R_0) \subseteq S' = (\Sigma', R_0 \cup E_0)$ with R_0 a ground convergent term rewriting system on T_Σ. The general idea is to complete first $R_0 \cup E_0$, yielding a ground convergent system R' on $T_{\Sigma'}$. Then one checks that whenever a rewrite rule, whose left and right-hand sides both belong to T_Σ, is added, then this rule is an inductive consequence of R_0. Bachmair has designed an unfailing ground completion procedure for consistency proofs in [1].

If the term rewriting system R associated with the specification is ground confluent, deciding sufficient completeness with respect to C is the same as checking that the normal form of all ground terms is a constructor term. If R preserves constructor terms, (i.e. for any rule $l \rightarrow r$ where l is a constructor term, r is also a constructor term), then it is equivalent to checking for inductive reducibility [11]. Deciding inductive reducibility can be done by using test sets. A constructive method for test sets is given by Kounalis in [13].

Ground confluence of the associated term rewriting system is required for proofs about enrichments. However, we do not always require consistency or sufficient completeness of enrichments. A specification that builds the integers modulo 2 by enriching a specification of integers is not consistent. A specification that builds integers with an infinity element by enriching a specification of integers is not sufficiently complete. Still, these kinds of construction can both be useful. Moreover, we do not really want to limit the transformation process to terminating programs. However, we are limited if we want to do automatic proofs about enrichments.

2 Program transformation

Dershowitz has shown how completion can be applied to the task of program synthesis from specifications in [7, 9]. The transformation process can be viewed as a *partial unfailing completion.*

Example 3 Let us take the well known example of the transformation of the specification of the function *reverse* in example 2 [7]. We want a more efficient implementation of *reverse*. In an attempt to find one, we enrich the specification with the definition of a new function motivated by a generalization of the right-hand side of equation 4.

$$h : list \times list \mapsto list$$
$$h(u,v) = append(reverse(u), v) \tag{5}$$

Overlaps between the right-hand side of equation 5 and the left-hand sides of equations 3 and 4 produce ordered critical pairs resulting in a direct definition of the function h:

$$h : list \times list \mapsto list$$
$$h(nil, v) = append(nil, v) \tag{6}$$
$$h(x :: xs, v) = append(append(reverse(xs), x :: nil), v) \tag{7}$$

This corresponds to applications of the instantiation law followed by an unfolding in the system of Burstall and Darlington [5]. The right-hand side of the equation 6 can be simplified using the definition of *append*:

$$h(nil, v) = v \tag{8}$$

The right-hand side of equation 7 can be simplified successively using the associativity of *append* given by equation 1, the definition of append, equation 2, and equation 5, oriented from right to left into:

$$h(x :: xs, v) = h(xs, x :: v)$$

This corresponds to applications of laws, unfoldings and finally a folding in the system of Burstall and Darlington. An overlap between the left-hand side of equation 2 and the right-hand side of equation 5 results in the equation:

$$reverse(x) = h(x, nil)$$

This overlap is another motivation for proposing equation 5. This completes the transformation of *reverse* using *append* into a tail recursive definition of *reverse* using only ::.

If we look at diverse examples, the heuristic is always the same: given a specification which defines a function f by equations, the first step consists of the introduction of a new function $h(x_1, \cdots, x_n) = e$, where e is chosen from the following heuristics:

- generalization of a subexpression e_f in the definition of f i.e. $e_f = \sigma(e)$ for some substitution σ so that e_f can be simplified into $\sigma(h(x_1, \cdots, x_n))$,
- a simple composition of functions in the definition of f and

- a tuple of subexpressions in the definition of f chosen from any of these heuristics.

Often, it happens that $f(x'_1, \cdots, x'_p)$ is a subexpression of e because the definition of f is recursive.

Overlaps between the left-hand side of the definition of h and the right-hand sides of one or more of the equations of f result in a direct definition of h by a set of equations d_h.

The second step consists in the simplification of the left-hand sides of d_h using equations of the original specification S of f and equations of an enrichment of S. Instances of e are simplified into instances of h.

If $f(x'_1, \cdots, x'_p)$ is a subexpression of e, it happens (mostly because the user has chosen e on purpose) that an instance of e can be simplified into $f(x'_1, \cdots, x'_n)$, resulting in a direct definition of f using h. In any case, because of the heuristics used to choose e, e_f can be simplified, resulting in a definition of f using h.

Let us consider another simple example to illustrate the tupling heuristic.

Example 4 The following specification (Σ, E) of integers:

$$\begin{aligned}
&sort\ Int \\
&ZERO\ :\ \mapsto Int \\
&S\ :\ Int \mapsto Int \\
&+\ :\ Int \times Int \mapsto Int \\
&*\ :\ Int \times Int \mapsto Int \\
&\forall x : Int.ZERO + x\ =\ x \\
&\forall x : Int.y : Int.S(x) + y\ =\ S(x + y) \\
&\forall x : Int.ZERO * x\ =\ ZERO \\
&\forall x : Int.y : Int.S(x) * y\ =\ x * y\ +\ y
\end{aligned}$$

is enriched with a definition of the function fib defining the n^{th} fibonacci number:

$$\begin{aligned}
&fib\ :\ Int \mapsto Int \\
&fib(ZERO)\ =\ ZERO && (9) \\
&\forall x : Int.fib(S(ZERO)\ =\ S(ZERO) && (10) \\
&\forall x : Int.fib(S(S(x)))\ =\ fib(S(x))\ +\ fib(x) && (11)
\end{aligned}$$

We will now generalize $fib(S(x))\ +\ fib(x)$ using a new function g by the tupling heuristic introducing as a new sort, pairs of integers:

$$\begin{aligned}
&sort\ :\ pair[elem] \\
&\langle -, - \rangle\ :\ elem \times elem \mapsto pair \\
&fst\ :\ pair \mapsto elem \\
&snd\ :\ pair \mapsto elem \\
&\forall x : elem.y : elem.fst(\langle x, y \rangle)\ =\ x \\
&\forall x : elem.y : elem.snd(\langle x, y \rangle)\ =\ y
\end{aligned}$$

We define g by:

$$\begin{aligned}
&g\ Int \mapsto Int \\
&\forall x : Int.g(x)\ =\ \langle fib(S(x), fib(x) \rangle && (12)
\end{aligned}$$

Overlaps between the left-hand side of the definition of g and the definitions of fst and snd result in:

$$fib(S(x)) = fst(g(x)) \tag{13}$$

$$fib(x) = snd(g(x)) \tag{14}$$

Equation 14 is a new definition of fib using g. Equation 11 is simplified into :

$$fib(S(S(x))) = fst(g(x)) + snd(g(x)) \tag{15}$$

Equation 12 is simplified into

$$\langle fst(g(x)), snd(g(x)) \rangle = g(x) \tag{16}$$

An overlap between 14 and 9, and an overlap between 13 and 10 results in:

$$fst(g(ZERO)) = S(ZERO)$$
$$snd(g(ZERO)) = ZERO$$

instantiating 16 into:

$$g(ZERO) = \langle S(ZERO), ZERO \rangle \tag{17}$$

Overlaps between 14, 13 and 15 result in:

$$fst(g(S(x))) = fst(g(x)) + snd(g(x))$$
$$snd(g(S(x))) = fst(g(x))$$

instantiating 16 into:

$$g(S(x)) = \langle fst(g(x)) + snd(g(x)), fst(g(x)) \rangle \tag{18}$$

Equations 14, 17, and 18 constitute a tail recursive definition of fib.

This transformation process is not restricted to simple and well known examples. The interested reader can look at the development of the Kwic example given in the appendix. Reddy gives very interesting examples in [19]. I will not address in this paper the question of the amelioration of the efficiency of a program by using this transformation process with the heuristics described above. I am only interested here in its correctness and its implementation using term-rewriting techniques.

2.1 Correctness of the transformation process

The transformation process consists primarily of that part of the unfailing completion process that I call a *partial unfailing completion.*

Definition 2 *Two specifications $S = (\Sigma, E)$ and $S' = (\Sigma, E')$ are equivalent if for any ground terms s and t, $s \longleftrightarrow^*_E t$ iff $s \longleftrightarrow^*_{E'} t$.*

In other words, S and S' have the same initial algebra. In the following, $t_1, \cdots, t_n$ are constructor terms. Recall that C is the set of constructors. Therefore, $T(C)$ is the set of ground constructor terms.

Definition 3 *Let $S_f = (\Sigma_f, E_f)$ be the specification defining the function f. The result of the transformation is a specification $S'_f = (\Sigma'_f, E'_f)$ specifying the same function f i.e. for all ground terms, $f(t_1, \cdots, t_n) \longleftrightarrow^*_{E_f} s$ if $f(t_1, \cdots, t_n) \longleftrightarrow^*_{E'_f} s$. In other words, S_f and S'_f are equivalent on the terms $T(C \cup \{f\}) \times T(C)$*

Proposition 1 *Let us call $S = (\Sigma, E)$ the enrichment of $S_f = (\Sigma_f, E_f)$ with a set of new function symbols Σ_h, their definitions E_h, and inductive consequences L of E. We have $\Sigma = \Sigma_f \cup \Sigma_h$ and $E = E_f \cup E_h \cup L$. Let $S' = (\Sigma, E')$ be the result of a partial unfailing completion of S. Then*

1. *The partial unfailing completion transforms S into an equivalent specification S'.*
2. *The transformation process transforms a specification $S_f = (\Sigma_f, E_f)$ of a function f into an equivalent specification $S'_f = (\Sigma'_f, E'_f)$ of the function f if*
   - *The E_C-equality (equality between constructors) is included into the E'_f-equality,*
   - *S is consistent with respect to the conctructors and*
   - *S'_f is a complete definition of f, i.e. for all ground terms $f(t_1, \cdots, t_n)$, there exists a constructor term s such that $f(t_1, \cdots, t_n) \longleftrightarrow^*_{E'_f} s$.*

Proof: The first result follows simply from the fact that partial unfailing completion does not modify the initial algebra. Considering the second result, the transformation process transform S_f into S'_f. First, S_f and S are equivalent because neither inductive consequences nor E_h modifies the initial algebra. Second, the partial unfailing completion does not modify the E-equality, thus $\longleftrightarrow^*_{E'_f} \subseteq \longleftrightarrow^*_E$. Let us consider a ground term $f(t_1, \cdots, t_n)$, and a constructor term s such that $f(t_1, \cdots, t_n) \longleftrightarrow^*_{E'_f} s$, then:

- $f(t_1, \cdots, t_n) \longleftrightarrow^*_E s$ by $\longleftrightarrow^*_{E'_f} \subseteq \longleftrightarrow^*_E$.
- Conversely, if $f(t_1, \cdots, t_n) \longleftrightarrow^*_{E_f} s$, then $f(t_1, \cdots, t_n) \longleftrightarrow^*_E s$ because S_f and S are equivalent. There exists a constructor term u such that $f(t_1, \cdots, t_n) \longleftrightarrow^*_{E'_f} u$. Therefore $f(t_1, \cdots, t_n) \longleftrightarrow^*_E u$ by $\longleftrightarrow^*_{E'_f} \subseteq \longleftrightarrow^*_E$. $u \longleftrightarrow^*_E s$ by transitivity of the E-equality. $u \longleftrightarrow^*_{E_C} s$ by consistency of E with respect to the conctructors. $u \longleftrightarrow^*_{E'_f} s$ because the E'_f-equality contains the E_C-equality. $f(t_1, \cdots, t_n) \longleftrightarrow^*_{E'_f} s$, by transitivity of the E-equality.

□

Proposition 2 *The transformation process preserves the consistency of a specification with respect to the conctructors.*

Proof: The partial unfailing completion does not modify the initial algebra.

□

Let us consider now the operational point of view. The theory is presented by a term rewriting system R for the specification (Σ, R). Computation of a term in $T(\Sigma)$ is done by rewriting. The operationally complete definition of a function f with a specification

(Σ_f, R_f) w.r.t C is when for all ground term $f(t_1, \cdots, t_n)$, there exists a constructor term s such that

$$f(t_1, \cdots, t_n) \rightarrow_R^* s$$

Operational and algebraic definitions coincide if R is ground convergent.

Proposition 3 *Let the specification (Σ_f, R_f) be an operationally complete definition of f consistent with respect to the conctructors. If it is transformed into the specification (Σ'_f, R'_f) which is an operationally complete definition of f, then the computations, i.e. the normal forms of a ground term $f(t_1, \cdots, t_n)$, by R and R' are E_C-equal.*

> **Proof:** A ground term $f(t_1, \cdots, t_n)$ has R_f-normal forms s and R'_f-normal forms u that are constructor terms. The R_f equality and the R'_f equality are contained into the $E = R_f \cup E_h \cup L$ equality where R_f is the set of rules of the new functions and L the set of the inductive consequences introduced for the transformation. Therefore, u and s are $(E, >)$-normal forms. Σ, E is consistent w.r.t C because the addition of L and E_h does not modify the consistency. $u \longleftrightarrow_{E_C}^* s$ by consistency of (Σ, E) with respect to the conctructors. □

All the above results are valid, if we replace the specification of the constructors (C, E_C) by any specification at any level of the hierarchic construction of the specification. The following result proved in [14] is interesting when we construct a specification hierarchically.

Proposition 4 *Let R_0 be a ground convergent term rewriting system, and let R_f be an enrichment of (Σ_0, R_0) with a complete definition of f w.r.t Σ_0, then R_f is ground convergent.*

As a consequence, when R_f is transformed into $R_0 \subseteq R'_f$, which gives a new complete definition of f w.r.t R_0, R'_f is ground convergent.

2.2 Implementation of the transformation process

The core of the system is the partial unfailing completion. Given a source term rewriting system R_f, we enrich the specification $S = (\Sigma_f, E_f)$ with a definition E_h of a new symbol h, like h in the example of *reverse* or like g in the example of *fibonacci*. This new definition is given in general by a unique equation $E_h : h(x_1, \cdots, x_n) = e$ where e is built following the diverse heuristics suggested above. Let us imagine how to organize the partial completion process.

1. *The system computes the ordered critical pairs between E_h and R_f*

   Let σ be the most general unifier of e with $f(t_1, \cdots, t_n)$, left-hand side of an equation $f(t_1, \cdots, t_n) = t$. Let $\sigma(e)$ be greater than the instance $\sigma(h(x_1, \cdots, x_n))$ so that the ordered critical pairs are equations $\sigma(h(x_1, \cdots, x_n)) = \sigma(t)$. If R_f contains complete definitions of every defined symbol, such equations contain a complete definition of h.

2. *The system processes simplifications*, then, the complete definition of f is simplified, and h must occur in the definition of f. The possible overlaps with E_h can give more than one possibility, as shown in the following example:

Example 5 Let R_f be a ground convergent system for a complete definition of factorial containing a definition of $+$ and $*$ on integers.

$$
\begin{aligned}
S(x) + y &\rightarrow S(x+y) \\
ZERO + x &\rightarrow x \\
S(x) * y &\rightarrow x * y + y \\
ZERO * x &\rightarrow ZERO \\
fact(S(x)) &\rightarrow S(x) * fact(x) && (19) \\
fact(ZERO) &\rightarrow S(ZERO) && (20)
\end{aligned}
$$

With a definition $h(x,u) = u * fact(x)$, the process will return the equations:

$$
\begin{aligned}
h(x, S(u)) &= h(x,u) + fact(x) && (21) \\
h(x, ZERO) &= ZERO && (22) \\
h(S(x), u) &= u * (h(x,x) + fact(x)) && (23) \\
h(ZERO, S(u)) &= x * S(ZERO) && (24)
\end{aligned}
$$

by overlapping $u * fact(x)$ on the definition of $*$ and on the definition of $fact$. The user may be disturbed by these two potential complete definitions of h.

3. *Additional inductive theorems can be added to help the transformation*

   Theorems are helpful

   (a) for simplifying all rules and equations

   (b) for deducting new inductive equations from the definition of the new symbol.

   Therefore, *we do not overlap the theorems with E_h and the consequences of E_h.* The system overlaps the new theorems only with E_f. Moreover, *some theorems can be used only for simplification and in this case they need not to be overlapped with E_f.* The user must indicate if the theorem must be overlapped or not. *New theorems can be provided by the user at the beginning of the whole process or during the process.* The separation of the transformation process into two steps as we illustrated here is totally artificial.

2.3 Limitations of partial completion

1. *The partial completion can loop* as every completion procedure can do. However, the user can always interrupt the process when getting a result that contains a new, hopefully better, complete definition of the function of interest.

   Example 6 Let us continue our example with the definition of factorial. First, we introduce the inductive theorem $x * S(ZERO) \rightarrow x$ for simplifying the right-hand side of equation 24 into

   $$h(ZERO, S(u)) = x \qquad (25)$$

   Second, we introduce the associativity of $*$ in an attempt to simplify the left-hand side $u*(h(x,x)+fact(x))$ of equation 23 and remove the occurrence of $fact(x)$. Assuming that $z*(u*fact(x))$ is greater than $(z*u)*fact(x)$, a superposition of the associativity on $h(x,u) = u * fact(x)$ generates the pair:

$$z * h(x,u) = h(x, z * u) \tag{26}$$

Assuming that $h(x,x) + fact(x)$ is greater than $h(x, S(x))$, the right-hand side of equation 23 is simplified, resulting in:

$$h(S(x), u) = h(x, u * S(x)) \tag{27}$$

Equations 25 and 27 give a tail recursive complete definition of h, but the other superpositions make the completion process continue indefinitely. One can notice that the process would work and be finite if superpositions were limited to being done only with the definition of $fact$ given by equations 19 and 20.

2. *The process can also fail to find the desired result, even if it exists, because of the inadequacy of the ordering.* This is the principal drawback of this method. Recursive path ordering [6] often does not work as well as polynomial interpretations [15]. Transformation orderings [3, 4] might be useful. Work remains to be done to find adequate orderings.

One way to resolve this can be to restrict the completion more severely. Let g be the function such that the overlaps between the definition of g and the definition of h must be done to find the new definition of h. Given the new function h, the function of interest f, the inductive consequences L, the means to orient equations of L, and the function g, the system or the user needs only to orient equational consequences.

The Focus system [18], which does not search for equational consequences, is even more restrictive. It superposes only g and h and simplifies by rewriting. Therefore, it does no completion at all. But this is sometimes too restrictive. For example, it generates this definition of $reverse$ as

$$reverse(nil) = nil$$
$$reverse(x :: xs) = h(xs, x :: nil)$$

but it does not generate the definition $reverse(x) = h(x, nil)$, although this last definition can always be proved by induction [20, 14] from the first one. The following example shows the weaknesses of the various choices.

Example 7 Let us go back to factorial. With the associativity of $*$ oriented as $x*(y*z) \to (x*y)*z$, the superposition between $u * fact(x))$ and $fact(S(x))$ is $u * fact(S(x))$ which has 3 distinct normal forms giving 3 definitions of $h(S(x), u)$ as:

1. $h(S(x), u) = u * (h(x,x) + fact(x))$
2. $h(S(x), u) = u * h(x, S(x))$
3. $h(S(x), u) = h(x, u * S(x))$

The third one gives a tail recursive definition. The partial completion will force the confluence to a unique normal form. If the ordering is well chosen (we noted above that this is the major drawback of the method), the third definition will be the normal-form. On another hand, without completion, you might get either the third definition with an outermost rewriting, or the first or the second ones with an innermost rewriting. The first definition is obtained by choosing to simplify first with the rule $S(x) * y) \to x * y + y$, and the second definition is obtained by choosing to simplify first with the definition of $h(x,u)$.

3 Conclusion

We choose to use a partial unfailing completion process as the central part of a transformation system. For that we use the toolkit of rewriting tools provided by ORME [16]. With this simple initial implementation we have tested the well-known examples and the kwic example given in the appendix. The kwic example is interesting because it requires 3 steps of transformation and therefore shows the potential for transformation of larger specifications by composition of individual transformation steps. With abilities to perform:

1. induction proofs [14, 20],

2. check of consistency [1, 9],

3. check of complete definition and sufficient completeness [11, 13],

one could check the main properties of the specifications and prove the inductive consequences that must be added to perform the transformation. The system must also extract the specification (Σ_f, R_f) from the specification (Σ', R') resulting from a transformation step, i.e. extract a complete definition of the function of interest f and, iteratively, complete definitions of functions that occur in the definition of f. For this purpose it needs a check of a complete definition.

All this might be extended to conditional specifications, i.e. a set of conditional equations. A conditional equation is an equation or an expression $e_1 \wedge \cdots \wedge e_n \Rightarrow e$ where $e_1, \cdots, e_n$ are equations called conditions and e is an equation. Conditional equations are very useful to express specifications. The function *filter* in the appendix might rather be expressed by:

$$\begin{aligned}
&filter(nil) = nil \\
&issig(x) \Rightarrow filter(cons(x, xs)) = filter(xs) \\
&not(issig(x)) \Rightarrow filter(cons(x, xs)) = cons(x, filter(xs))
\end{aligned}$$

They are also very useful to express conditional properties that might help the transformation.

Acknowledgements: We thank U.S. Reddy for giving us a copy of the Focus System, P. Lescanne for the tools provided by ORME, R. Kieburtz for the idea of the kwic example, B. Vance, J. Hook, R. Kieburtz, C. Kirchner and P. Lescanne for their support and comments.

References

[1] L. Bachmair. Proofs methods for equational theories. PhD thesis, University of Illinois, Urbana-Champaign, 1987. Revised version, August 1988.

[2] L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In *Proceedings of the colloquium on the resolution of Equations in Algebraic Structures*, 1987.

[3] F. Bellegarde and P. Lescanne. Transformation Orderings. In *12th Colloquium on Trees in Algebra and Programming*, TAPSOFT, pages 69-80, Springer Verlag, Lecture Notes in Computer Science 249, 1987.

[4] F. Bellegarde and P. Lescanne. Termination by Completion, Technical Report CRIN 90-R-028, 1990.

[5] R. M. Burstall and J. Darlington. A Transformation System For Developing Recursive Programs. *Journal of the Association for Computing Machinery*, 24, pages 44-67, 1977.

[6] N. Dershowitz. Termination. In *Proceedings of the first Conference on Rewriting Techniques and Applications*, Springer Verlag, Lecture Notes in Computer Science 202, pages 180-224, Dijon, France, 1985.

[7] N. Dershowitz. Synthesis By Completion. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 208-214, Los Angeles, 1985.

[8] N. Dershowitz. Computing with rewrite systems. *Information and Control*, 65(2/3):122-157, 1985.

[9] N. Dershowitz. Completion and its Applications. *Resolution of Equations in Algebraic Structures,* Academic Press, New York, 1988.

[10] J. Goguen and C. Kirchner and H. Kirchner and A. Megrelis and J. Meseguer and T. Winkler. An introduction to OBJ-3. In *Proceedings of the 1rst Intern. Workshop on Conditional Term Rewriting Systems,* Lecture Notes in Computer Science 308, 1988.

[11] D. Kapur, P. Narendran, and H. Zhang. On sufficient completeness and related properties of term rewriting systems. *Acta Informatica,* 24:395-415, 1987.

[12] D. E. Knuth and P. B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract algebra*, pages 263-297, Pergamon Press, Oxford, U. K.,1970.

[13] E. Kounalis, Testing for Inductive (CO)-Reducibility. In *Proceedings of the 15th International Colloquium on Trees in Algebra and Programming,* Lecture Notes in Computer Science 431, pages 221-238, 1990.

[14] E. Kounalis, M. Rusinowitch. Mechanizing Inductive Reasoning. In *Proceedings of the eight National Conference on Artificial Intelligence*, AAAI-90, 1990.

[15] D.S. Lankford. On proving term rewriting systems are Noetherian, *Memo MTP-3,* Mathematic Department, Louisiana Tech. University, Ruston, LA, May 1979. (Revised October 1979).

[16] P. Lescanne, Completion Procedures as Transition Rules + Control:ORME. In *2nd Intern. Workshop Algebraic and Logic Programming,* Lecture Notes in Computer Science, 1990.

[17] M. O'Donnell. *Equational Logic as a Programming Language. Foundation of Computing,* MIT Press, 1985.

[18] U. S. Reddy. Transformational derivation of programs using the Focus system. In *Symposium Practical Software Development Environments*, pages 163-172, ACM, December 1988.

[19] U. S. Reddy, Formal methods in transformational derivation of programs. In Proceedings of the ACM Intern. Workshop on Automatic Software Design, AAAI, 1990.

[20] U. S. Reddy, Term Rewriting Induction. In *Proceedings of the Conference of Automated Deduction,* 1990.

4 Appendix: Kwic example

The problem is to produce, from a list of titles, a list of the cyclic permutations of the original titles such that we retain only those permutations that begin with a key word.

4.1 Specification

Here we show the construction of the specification by successive enrichments.

Let us represent a title by a list of words. Our input is a list of titles. We can use a specification of lists.

$$
\begin{aligned}
&sort\ list[elem] \\
&nil\ : \mapsto list \\
&::\ :\ elem \times list \mapsto list \\
&append\ :\ list \times list \mapsto list \\
&\forall x : list.append(nil, x) \ = \ x \\
&\forall x : elem.xs, y : list.append((x :: xs), y) \ = \ x :: append(xs, y)
\end{aligned}
$$

We enrich the specification with the definition of a function *rotations* to get an elementary cyclic permutation.

$$
\begin{aligned}
&\forall x : list[word].rotate(nil) \ = \ nil \\
&\forall x : word.xs : list.rotate(x :: xs) \ = \ append(xs, x :: nil)
\end{aligned}
$$

Now we want to iterate the function *rotate* to get the permutations of a title. The title itself can be used to control the iteration. The function *all* returns the complete list of cyclic permutations of a title.

$$
\begin{aligned}
&\forall x : list[word].all(x) = repeat(x, x) \\
&\forall x : list[word].repeat(x, nil) = nil \\
&\forall x : word.u, xs : list[word].repeat(u, x :: xs) = u :: repeat(rotate(u), xs)
\end{aligned}
$$

We can now get the permutations of all titles by a function *concall*:

$$
\begin{aligned}
&concall(nil) = nil \\
&\forall x : list[word].xs : list[list[word]].concall(x :: xs) = append(all(x), concall(xs))
\end{aligned}
$$

The list of permutations can be filtered to extract the significant titles. The permutations whose initial word belongs to the set of insignificant words can be dropped. A predicate *issig* checks if the permutation is kept. *To specify filter, we use a ternary conditional operator cond.*

$$
\begin{aligned}
&filter(nil) = nil \\
&\forall x : list[word].xs : list[list[word]]. \\
&filter(x :: xs) = cond(issig(x), x :: filter(xs), filter(xs))
\end{aligned}
$$

Finally, we get the desired result by:

$$
\forall x : list[list[word]].sigperm(x) = filter(concall(x))
$$

4.2 First transformation step

We first transform the definition of *sigperm*:

$$\begin{aligned}
&sigperm(nil) = nil \\
&sigperm(x :: xs) = filter(append(repeat(x, x), concall(xs))
\end{aligned}$$

We introduce an inductive theorem:

$$filter(append(x, y)) = append(filter(x), filter(y))$$

It allows us to simplify *sigperm* into:

$$sigperm(x :: xs) = append(filter(repeat(x, x)), sigperm(xs))$$

The complete definition of *sigperm* is now:

$$\begin{aligned}
&append(nil, x) = x \\
&append(x :: xs, y) = x :: append(xs, y) \\
&rotate(nil) = nil \\
&rotate(x :: xs) = append(xs, x :: nil) \\
&repeat(x, nil) = nil \\
&repeat(u, x :: xs) = u :: repeat(rotate(u), xs) \\
&filter(nil) = nil \\
&filter(x :: xs) = cond(issig(x), x :: filter(xs), filter(xs)) \\
&sigperm(nil) = nil \\
&sigperm(x :: xs) = append(filter(repeat(x, x)), sigperm(xs))
\end{aligned}$$

4.3 Second transformation step

We are now interested in modifying the composition $filter(repeat(x, x))$ in the definition of *sigperm*. We introduce a new definition:

$$sigrot(x, y) = filter(repeat(x, y))$$

and the transformation process gives the equations:

$$\begin{aligned}
&sigperm(x :: xs) = append(sigrot(x, x), concfil(xs)) \\
&sigrot(x, nil) = nil \\
&sigrot(u, x :: xs)) = cond(issig(u), u :: sigrot(rotate(u), xs), sigrot(rotate(u), xs))
\end{aligned}$$

The complete definition of *sigperm* is now:

$$\begin{aligned}
&append(nil, x) = x \\
&append(x :: xs, y) = x :: append(xs, y) \\
&rotate(nil) = nil \\
&rotate(x :: xs) = append(xs, x :: nil)
\end{aligned}$$

$$sigperm(nil) = nil$$
$$sigperm(x :: xs) = append(sigrot(x, x), sigperm(xs))$$
$$sigrot(x, nil) = nil$$
$$sigrot(u, x :: xs)) = cond(issig(u), u :: sigrot(rotate(u), xs), sigrot(rotate(u), xs))$$

4.4 Third transformation step

Our objective is to get rid of the costly occurrences of *append* in *sigperm*. We introduce the new definition:

$$sr(x, y, u) = append(sigrot(x, y), u)$$

and the theorem:

$$append(cond(u, x, y), z) = cond(u, append(x, z), append(y, z))$$

the transformation process returns:

$$sr(x, nil, u) = u$$
$$sr(x1, x :: xs, u) = cond(issig(x), x :: sr(rotate(x1), xs, u), sr(rotate(x1), xs, u))$$
$$concfil(x :: xs) = sr(x, x, concfil(xs))$$

The complete definition of *sigperm* is now:

$$append(nil, x) = x$$
$$append(x :: xs, y) = x :: append(xs, y)$$
$$rotate(nil) = nil$$
$$rotate(x :: xs) = append(xs, x :: nil)$$
$$sigperm(nil) = nil$$
$$sigperm(x :: xs) = sr(x, x, concfil(xs))$$
$$sr(x, nil, u) = u$$
$$sr(x1, x :: xs, u) = cond(issig(x), x :: sr(rotate(x1), xs, u), sr(rotate(x1), xs, u))$$