ISRN SICS-R–94/02–SE

Unfolding of Programs with Nondeterminism

Björn Lisper

SICS research report
R94:02
ISSN 0283-3638

R94:02 0283-3638

Unfolding of Programs with Nondeterminism

by

Björn Lisper
Formal Design Techniques; February 2, 1994
lisper@it.kth.se

Swedish Institute of Computer Science
Box 1263
S-164 28 KISTA
Sweden

Abstract

In [13], some results were proved regarding properties of unfolding of purely functional programs. Especially, a theorem was shown that relates the termination of symbolic evaluation of a "less instantiated" term relative to the termination of a "more instantiated" term. An application is partial evaluation, where unfolding of function definitions is frequently performed to enable further simplifications of the resulting specialized program. The unfolding must then be kept under control to ensure that the partial evaluation terminates. In this paper, we extend the termination result from purely functional programs programs with nondeterministic operations. We give an operational semantics where the behaviour of operators is defined through rewrite rules: nondeterminism then occurs when the resulting term rewriting system is nonconfluent. For the confluent part, the previous termination results carry over. It is, however, not guaranteed in general that the resulting unfolded program has the same semantics as the original program. We give conditions on the rewrite rules that guarantee that both versions have the same semantics, and we show that they apply to a nontrivial class of nondeterministic languages.

Keywords: nondeterminism, unfolding, recursive programs, term rewriting.

1 Introduction

A class of source-to-source optimizing transformations of purely functional programs is based on *unfolding*, where a function definition is substituted into the position of a function call and actual parameters are substituted for formal parameters. The result can then be manipulated in various ways. One possibility is to apply algebraic rewritings followed by a *fold* forming a new recursive definition, as in Burstall's and Darlington's classical fold-unfold-transformations [6, 9]. Another possibility is to take advantage of the possible simplifications that often occur when an unfolded function definition occurs in a calling environment, whith actual arguments instead of formal arguments. This is a common transformation in *partial evaluation* [11].

Partial evaluation refers to the technique of how to derive a specialized instance of a general function definition. In the type case, a recursive function definition

$$f(x, y) = e$$

is given. In some context, f then appears with partially known arguments, say as $f(x, c)$ where c is a known constant. Instead of calling the general function f applied to the arguments x and c, it may then pay off to derive a specialized function $f_c(x)$ in only one argument and replace the call $f(x, c)$ with a call $f_c(x)$. Typically, the increased information about the arguments ($y = c$) will result in a more efficient program.

A classical example of unfolding in partial evaluation is given below. Consider the following recursive definition of the x^n-function:

$$f(x, n) \leftarrow \mathit{if}(n = 0, 1, x \cdot f(x, n-1)).$$

Say, now, that a call $f(x, 3)$ occurs in some context. We can derive a definition of a specialized version $f_3(x)$ as follows:

$$\begin{aligned} f_3(x) &\leftarrow \mathit{if}(3 = 0, 1, x \cdot f(x, 3-1)) \\ &\leftarrow x \cdot f(x, 2) \\ &\leftarrow x \cdot \mathit{if}(2 = 0, 1, x \cdot f(x, 2-1)) \\ &\leftarrow x \cdot (x \cdot f(x, 1)) \\ &\vdots \\ &\leftarrow x \cdot (x \cdot (x \cdot 1)). \end{aligned}$$

A call to f_3 will use a total of three arithmetical operations, probably a great deal less work than performing the original call $f(x, 3)$, which involves costly operations like evaluating conditionals and performing recursive calls.

The x^n-example above is actually quite extreme: here, unfolding of recursive calls are carried out to the "bitter end", until *no* recursive calls remain. We may call this strategy "total unfolding".

In the example above, total unfolding terminated in three steps. It may, however, fail to terminate. A simple example is unfolding of $f(3, n)$ rather than $f(x, 3)$:

$$\begin{aligned} f_3(n) &\leftarrow \mathit{if}(n = 0, 1, 3 \cdot f(3, n-1)) \\ &\leftarrow \mathit{if}(n = 0, 1, 3 \cdot \mathit{if}(n - 1 = 0, 1, 3 \cdot f(3, n-2))) \\ &\vdots \end{aligned}$$

Clearly, total unfolding will not terminate here.

In [13], a formal model for unfolding is developed, along the lines of recursive applicative program schemes [8, 14]. The model is used to prove a theorem that under certain conditions relates termination of symbolic unfolding to the termination of a call to the original function. It is also shown that unfolding preserves the semantics. A short review of the results in [13] is given in section 3.

All the above holds for purely functional, deterministic programs. The purpose of this paper is to extend the previous results for unfolding to programs with nondeterminism. Two obstacles can immediately be identified: first nondeterminism must be controlled in some way during the unfolding, since uninhibited use may cause the resulting specialized program to be "too special". If a pure nondeterministic choice is evaluated during unfolding, then that choice will be have been done once and for all in the resulting function. This is sometimes not the intended result, since there can be an implicit assumption of fairness with regard to the different choices. Even worse, nondeterminism can break the relative termination between symbolic and "full" evaluation: informally, a choice can go one way during unfolding and another way during full evaluation, which in turn can make a condition come out differently in the respective cases.

Still, there are definitely cases when it is interesting to unfold programs with nondeterminism. A practical example is programs with *processes*: processes are important for concurrent programming but programs with processes will as a rule have nondeterminism. Unfolding will typically result in programs with simpler control structure: for processes, for instance, this can mean that processes being dynamically created in the general program can be allocated statically in the specialized program, with a corresponding reduction in dynamic overhead, increased possibility for static process allocation, etc. Consider the following example. We assume an applicative language with processes: especially, there is a parallel composition "|" and an "empty process" "**0**". The following is a recursive definition of a "fork" operation that spawns n copies of a process P:

$$fork(P,n) \leftarrow\ if(n=0,\mathbf{0},P|fork(P,n-1)).$$

Clearly, we can unfold a call $fork(P,3)$:

$$\begin{array}{rcl} fork_3(P) & \leftarrow & if(3=0,\mathbf{0},P|fork(P,3-1)) \\ & \leftarrow & P|fork(P,2) \\ & \vdots & \\ & \leftarrow & P|(P|(P|\mathbf{0})). \end{array}$$

This unfolding terminates in exactly the same manner as the unfolding of $f(x,3)$ above: this is not surprising, since the definitions of *fork* and f actually are schematically equivalent.

This paper is organized as follows. In section 3 we give a short summary of the results in the purely functional case: operators are modeled by rewrite rules that allow a confluent rewrite strategy. In section 4 the purely functional evaluation model is extended to include nondeterministic operators: these are modeled by a term rewriting system that is nonconfluent. It is shown how to restrict unfolding so the relative termination theorem still holds. The issue of *preservation of semantics* remains, however: even when termination is ensured it is not guaranteed, for nonconfluent rewriting systems in general, that the unfolded program will be able to produce exactly the same results as the original program. We give conditions on the rewrite rules that guarantee that both versions have the same semantics, and we show that they apply to a nontrivial class of nondeterministic languages. In section 5, finally, we wrap up the results and point at some interesting topics for further research.

2 Preliminaries

In this paper we consider terms. These are built from variables, constants and possibly with λ-abstractions that bind variables. A *closed term* has no free variables. Often there is some typing discipline that puts syntactic restrictions on the terms under consideration: it is for instance common to have a simple type system with *base types* from a set B and *function types* $\tau \to \tau'$ where τ, τ' are types.

For a simple type system, a constant of type $\tau_1 \times \cdots \times \tau_n \to \tau$, where $\tau_1, \ldots, \tau_n, \tau$ are base types, is called a *first order function symbol*. *First order terms* are built from constants and variables of base type, and then inductively by applying first order function symbols to first order terms. Terms which are not first order are called *higher order*. Such a term contains λ-abstractions, or variables or constants (besides first order function symbols) that are not of base type.

A *substitution* is a type-preserving partial function from variables to terms. The *domain* of a substitution σ, $dom(\sigma)$, is the set of all variables x for which $\sigma(x)$ is defined. The *range* of σ, $rg(\sigma)$, is the set of all free variables that occur in any $\sigma(x)$. Substitutions are naturally extended to general terms. *Composition* of substitutions is defined as $\sigma_1\sigma_2 = \{\, x \leftarrow \sigma_1(\sigma_2(x)) \mid x \in dom(\sigma_2)\,\} \cup \sigma_1|_{dom(\sigma_1)\setminus dom(\sigma_2)}$ ("$|$" denotes restriction of function). It has the property that $\sigma_1\sigma_2(e) = \sigma_1(\sigma_2(e))$ always.

We will use *reduction systems* (sometimes called *abstract reduction systems*) to give semantics of operators. For completeness, we give the basic definitions. Also see [10] or [12]. Let $\to$, $\to_1$, $\to_2$ be binary relations over some set A. $\to^*$ denotes the transitive-reflexive closure of $\to$. Then:

1. $a \to_1 \circ \to_2 b$ iff there exists a c such that $a \to_1 c \to_2 b$.

2. $\to$ is *terminating* (or *noetherian*) if for any $a \in A$ there is no infinite chain $a \to a_1 \to a_2 \to \cdots$.

3. $a \in A$ is a *normal form* (w.r.t. $\to$), or $\to$*-nf*, if there is no $b \in A$ such that $a \to b$. We then say that *a is a normal form to c* if $c \to^* a$, and we denote this relation by $c \to^! a$.

4. $\to_1$ and $\to_2$ *commute* if $\forall a, b, c \in A \exists d \in A[(a \to_1^* b \wedge a \to_2^* c) \implies (b_2 \to_2^* d \wedge c \to_1^* d)]$.

5. $\to$ is *confluent* (or *Church-Rosser*, or *self-commuting*) if $\forall a, b, c \in A \exists d \in A[(a \to^* b \wedge a \to^* c) \implies (b \to^* d \wedge c \to^* d)]$.

6. $\to$ has the *unique normal form property* if any element has at most one normal form.

A confluent relation has the unique normal form property. An *abstract reduction strategy* w.r.t. $\to$ is a function F such that $F(a) = a$ if F is a normal form, and $a \to F(a)$ otherwise. Confluence and termination is defined similarly to the same concepts for relations, see [13] for the details (also cf. [3, p. 351]).

We are especially interested in reduction systems over terms. In order to define them we need the concepts of *position* and *replacement*. We define a position in a term to be a sequence of nonnegative integers, where Λ denotes the empty sequence. For any term e we define its *set of positions* $Pos(e)$ and the *subterm at position* p, e/p, as:

- If $e = x$ or $e = c$, then $Pos(e) = \{\Lambda\}$ and $e/\Lambda = e$.
- If $e = \lambda x.e'$, then $Pos(e) = \{\Lambda\} \cup \{\, 0p \mid p \in Pos(e')\,\}$, $e/\Lambda = e$ and $e/(0p) = e'/p$.

- If $e = e_0(e_1 \ldots e_n)$, then $Pos(e) = \{\Lambda\} \cup \{ ip \mid 0 \leq i \leq n, p \in Pos(e_i) \}$, $e/\Lambda = e$ and $e/(ip) = e_i/p$.

Furthermore, we define $FPos(e) = \{ p \mid p \in Pos(e) \wedge e/p \neq x \}$ (where x is any variable). p is a *prefix* of p', or $p \leq p'$, if there exists an u such that $p' = pu$. If neither $p \leq p'$ or $p' \leq p$, then p and p' are *disjoint.*

The *replacement in* e *with* e' *at* p, $e[p \leftarrow e']$, is defined by:

- $e[\Lambda \leftarrow e'] = \Lambda$.
- $(\lambda x.e)[0p \leftarrow e'] = \lambda x.(e[p \leftarrow e'])$.
- $e_0(e_1 \ldots e_n)[ip \leftarrow e'] = e_0(e_1 \ldots e_i[p \leftarrow e'] \ldots e_n)$.

Definition 1 *A* term rewriting system *is a set R of pairs of terms, where s is not a single variable, s and t have the same type and $(s,t) \in R \implies varset(s) \supseteq varset(t)$. The* reduction relation $\rightarrow_R$ *associated with R is the finest relation containing R that is closed under substitution and replacement, i.e.*

1. $t \rightarrow_R t' \implies \sigma(t) \rightarrow_R \sigma(t')$ *for any substitution* σ,
2. $t \rightarrow_R t' \implies t''[p \leftarrow t] \rightarrow_R t''[p \leftarrow t']$ *for any term t'' and position* $p \in Pos(t'')$.

Each pair in R is called a rewrite rule. *A rewrite rule (s,t) is* erasing *if $varset(s) \supset varset(t)$, otherwise* non-erasing.

Sometimes we will consider pure term rewriting systems and sometimes systems that possibly contain other types of reductions, like β-reduction. We will call such more general systems "term reduction systems".

Finally, we need the notion of *residual* (cf. [4]). Formally, we express this as a relation between "old" and "new" positions. This relation is a function of the expression e to be rewritten, the position p of the redex, and the matching rewrite rule (s,t). For any position $p' \in Pos(e)$, the relation $Res(e,p,s,t)$ is given by:

- If $p \not\leq p'$, then p' $Res(e,p,s,t)$ p'.
- If $p' = puw$, $s/u = x$ and $t/v = x$ for some variable x, then p' $Res(e,p,s,t)$ pvw.

For any set of positions $P \subseteq Pos(e)$, we define $Res(P,e,p,s,t) = \{ p'' \mid \exists p' \in P \colon p'\ Res(e,p,s,t)\ p'' \}$. Whenever the meaning is clear from the context we will write $Res(P)$ rather than $Res(P,e,p,s,t)$. We will also use $Res(P)$ to denote the set of residuals originating from P after several rewritings, i.e. in an expression e' where $e \rightarrow_R^* e'$ (with the obvious definition for the case $e \rightarrow_R^0 e'$).

3 Unfolding of Deterministic Programs

The following is a review of the models and results in [13]. First, we give an evaluation model for recursive programs.

Definition 2 *A* recursive program π *is a finite substitution where $rg(\pi) \subseteq dom(\pi)$, no variable in $dom(\pi)$ occurs in any rule of ϵ, and any $f \in dom(\pi)$ has the same type as $\pi(f)$.*

If π is a recursive program, then we will refer to each $f \in dom(\pi)$ as a *defined function* of π, and to $\pi(f)$ as the *definition* of f.

The *full substitution evaluation rule* is known to be a fixpoint computation rule [7], i.e. it terminates with the correct answer exactly when the function that is least fixpoint solution of the recursive definition is defined. It can be informally described as follows. In order to compute a function call $f(a_1 \ldots a_n)$, do the following:

1. Replace concurrently all occurrences of defined functions with their definitions.
2. Simplify the result as far as possible.
3. Repeat steps 1 and 2 until termination, i.e. until a closed term remains (representing a value in the domain we are computing in).

The simplification phase is modeled by a term rewriting system: the rewrite rules can be seen as operational definitions of the operators in the language under consideration. The term rewriting system is divided into two parts: ν that contains only *non-erasing* rules, and ϵ that can contain *erasing* rules. Typically, ν will contain rules like δ-reductions of closed terms to constants, e.g. $2+3 \rightarrow_\nu 5$, whereas ϵ contains rules for nonstrict operations. For instance, the following two ϵ-rules give operational semantics to the usual *if*-function:

$$\begin{aligned} \mathit{if}(\mathbf{true}, x, y) &\rightarrow_\epsilon x \\ \mathit{if}(\mathbf{false}, x, y) &\rightarrow_\epsilon y. \end{aligned}$$

"Parallel" rules can also be introduced, like "parallel if":

$$\mathit{if}(x, y, y) \rightarrow_\epsilon y.$$

The term rewriting systems ν and ϵ give rise to the reduction relations $\rightarrow_\nu$ and $\rightarrow_\epsilon$ on the terms under consideration.

Simplification also includes the usual β-reduction of λ-terms. Thus, there is also a reduction relation $\rightarrow_\beta$. Of particular interest is the combined reduction relation $\rightarrow_{\beta\epsilon\nu} = \rightarrow_\nu \cup \rightarrow_\beta \cup \rightarrow_\epsilon$. We assume that there is a confluent and terminating (one-step) reduction strategy F w.r.t. $\rightarrow_{\beta\epsilon\nu}$. Then, for each term t, repeated application of F will eventually rewrite t into a normal form F^*t. F^* is then a well-defined tramsformation from terms to terms, and we take it as our formal definition of the simplification phase in the full substitution rule.

Given ν, ϵ and a set of terms under consideration, it is of course interesting to establish confluence and termination of F: see the discussion in [13]. In all "reasonable" cases, when deterministic operations are modeled, F will have these properties.

One full step of the full substitution rule can now be expressed as the function $F^*\pi$ on terms. Repeated application of $F^*\pi$ models several steps of the full substitution rule. We define termination as $(F^*\pi)^n$ being free of π-defined functions for some n:

Definition 3 *Let e be a term. Then, if for some $n > 0$ it holds that $(F^*\pi)^n e$ is free from variables in $dom(\pi)$, we define $(F^*\pi)^* e = (F^*\pi)^n e$. The evaluation of e* terminates *if $(F^*\pi)^* e$ is defined.*

The nice thing with the full substitution evaluation rule as defined above is that it generalizes directly to symbolic evaluation. Especially, it models total unfolding in the symbolic case. Thus, we can use it to compare symbolic and "full" evaluation. We must then have some notion of "less" and "more" instantiated terms. A relation "$\geq_X$" (or rather: a family of relation, for sets of variables X) formalizes this: essentially $e \geq_X e'$ says that e is less instantiated than e', but they agree on the variables in X. Another concept that is needed is that of *potential X-match*: essentially, e has a potential X-match if there is a way to instantiate the variables outside X so that an erasing reduction can occur. See [13] for the formal definitions. It is then possible to prove the following:

Theorem 1 *(Relative termination of deterministic unfolding) If $dom(\pi) \subseteq X$, and if for all $n > 0$ holds that $(F^*\pi)^n e$ has no potential X-match, then $e \geq_X e'$ implies that $(F^*\pi)^* e$ is defined whenever $(F^*\pi)^* e'$ is defined.*

Not surprisingly, confluence and termination of F is crucial for proving this theorem (see [13] for the details). Consider now the classical pattern for partial evaluation: a partially instantiated function call $f(x, c)$, where c is a constant and x is a variable, relative to a totally instantiated function call $f(b, c)$ where also b is a constant. Whenever f is defined by π, where $dom(\pi) \subseteq X$ and $x \notin X$, it holds that $f(b, x) \geq_X f(b, c)$. Theorem 1 can then be applied.

Preservation of semantics of unfolding, finally, can be formulated as follows: evaluating an unfolded term, with free variables instantiated, should yield the same result as evaluating the original term with the free variables instantiated in the same way. Instantiation is done by a substitution θ where $dom(\theta) \cap dom(\pi) = \emptyset$ (i.e. θ can only instatiate free variables, not any defined functions). We also assume that $dom(\theta) \cap dom(\pi) = dom(\theta) \cap rg(\pi) = \emptyset$ (θ does not introduce any calls to functions in π): it then holds that $\theta\pi = \pi\theta$. Under these circumstances, the following result holds:

Theorem 2 *(Preservation of semantics of deterministic unfolding)*

$$(F^*\pi)^*\theta(F^*\pi)^* e = (F^*\pi)^*\theta e$$

whenever $(F^*\pi)^*$ is defined for e and θe.

4 Nondeterminism

Nondeterministic operations are often very naturally defined by rewrite rules. A simple example is pure nondeterministic choice "+", that can be defined by the rewrite rules

$$\begin{aligned} x + y &\rightarrow x \\ x + y &\rightarrow y. \end{aligned}$$

However, introduction of rules like the above will destroy confluence. This is exactly what is expected, since the spirit of nondetermininism is that several different answers are possible, whereas confluence always forces a single result. Put in another way, nondeterminism *is* nonconfluence.

Another consequence is that reduction *strategies* are not appropriate anymore, since they are by definition deterministic. So from now on we will give the semantics of operations by term reduction system R rather than a reduction strategy w.r.t. R. R can possibly include β-reduction: we will explicitly point out when this is not allowed.

When $\rightarrow_R$ is terminating, it is still appropriate to think of $\rightarrow_R$ as a simplification. The difference is that there now may be several normal forms. For any term t, we can take the *set* of its normal forms as the result of the simplification. In line with the earlier notation, we define

$$R^*(t) = \{\, t' \mid t \rightarrow^!_R t' \,\}$$

We can now extend the full substitution rule to an operational semantics for nondeterministic recursive programs. We assume that $\rightarrow_R$ is terminating, so R^* is always well-defined. For any set of terms E, we define $\pi E = \{\, \pi e \mid e \in E \,\}$. We also extend R^* to sets of terms, viz. $R^* E = \bigcup_{e \in E} R^*(e)$.

Definition 4 *Let E be a set of terms. Then, if for some $n > 0$ it holds that each element in $(R^*\pi)^n E$ is free from variables in $dom(\pi)$, we define $(R^*\pi)^* E = (R^*\pi)^n E$. The evaluation of E* terminates *if $(R^*\pi)^* E$ is defined.*

When $\rightarrow_R$ is both confluent and terminating, we will always have $(R^*\pi)^*\{e\} = \{(R^*\pi)^* e\}$ whenever defined. Furthermore, the termination theorem from section 3 carries over directly. When $\rightarrow_R$ is only terminating and not confluent, the theorem will however *not* hold in general. This is in line with the informal observation that a nondeterministic (nonconfluent) reduction may go one way at "compile-time" and another way at "runtime". Another problem that arises when nondeterministic rewrite rules are used at compile-time is that the semantics of the program can be changed: during unfolding, an expression

$$x + y$$

will then evaluate to either x or y. Then, the resulting unfolded program will either always pick x or always pick y, whereas the original program has the full choice $x+y$. With semantics according to definition 4 they will *not* be semantically equivalent in general.

The only reasonable approach to unfolding of programs with nondeterminism is thus to perform only deterministic reductions during symbolic unfolding. The issue then arises whether the result, when terminating, is guaranteed to be semantically equivalent with the original function. It turns out that this is not always the case. However, conditions on the rewrite rules can be established that will guarantee equivalence. We will now formalize the problem and prove some conditions for semantic equivalence.

Let us assume that we can factor R as $R = D \cup N$, where $\rightarrow_D$ is confluent and $\rightarrow_N$ is possibly not. For any term t, we denote its (unique) normal form w.r.t. $\rightarrow_D$ by D^*t. As for the fully deterministic case, we define the result of "deterministic unfolding" of a term t as $(D^*\pi)^*t = (D^*\pi)^n t$ if $(D^*\pi)^n t$ is free from functions defined by π. Preservation of semantics of for unfolding can now be formulated as in the deterministic case (we define it for single terms, but the definition can be trivially extended to sets of terms):

Definition 5 *The unfolding of t* preserves the semantics, *if $(R^*\pi)^*\theta(D^*\pi)^*t = (R^*\pi)^*\theta t$.*

Intuitively, *commutation* between $\rightarrow_D$ and $\rightarrow_N$ is essential for preservation of semantics for unfolding with deterministic reduction. Commutation simply says that if several different kinds of simplifications are possible, it does not matter in which order they are made. It should then be allright to perform only deterministic reductions at "compile-time" and save the nondeterministic to "runtime". The following well-known result is useful for establishing commutativity:

Proposition 1 *If the term rewriting systems R_1 and R_2 are orthogonal to each other (no overlap between a rule in R_1 and a rule in R_2) and left-linear, then $\rightarrow_{R_1}$ and $\rightarrow_{R_2}$ commute.*

Proof. See [15], [14], or the survey [12]. ∎

$$\begin{tikzcd}
a_0 \arrow[r, "N"', "*"] \arrow[d, "D", "*"'] & a_1 \arrow[r, "D"', "*"] \arrow[d, "D", "*"'] & a_2 \arrow[r, dotted, dash] \arrow[d, "D", "*"'] & {} \\
b_0 \arrow[r, "N"', "*"] & b_1 \arrow[r, "D"', "*"] & b_2 \arrow[r, dotted, dash] & {}
\end{tikzcd}$$

Figure 1: Diagram chase to prove commutation of $\rightarrow_D$ and $\rightarrow_R = \rightarrow_{D \cup N}$.

Note that if $\rightarrow_N$ is nonconfluent and $\rightarrow_D$ has a non-left-linear rule they will typically not commute. To see why, consider the following example. Let D be defined by

$$\mathit{if}(y, x, x) \rightarrow_D x$$

and let N be given by

$$\begin{aligned} x + y &\rightarrow_N x \\ x + y &\rightarrow_N y. \end{aligned}$$

Then, $\mathit{if}(y, x + z, x + z) \rightarrow_D x + z$ and $\mathit{if}(y, x + z, x + z) \rightarrow_N^* \mathit{if}(y, x, z)$. If x and z are distinct normal forms w.r.t. N, then the only N-reducts of $x + z$ are x and z, but there is no D-rule that rewrites $\mathit{if}(y, x, z)$. Thus, if a nonlinear rule is used in the simplification phase, there is a risk that some nondeterminism is "cancelled" and the unfolded version of the function has strictly less normal forms.

Let us now prove some results regarding the preservation of semantics. Again, we assume an instantiating substitution θ such that $\theta\pi = \pi\theta$. Interestingly, it turns out that commutativity of $\rightarrow_D$ and $\rightarrow_N$ is only enough to guarantee a "weak" form of semantics-preservation: we will formulate this concept below, prove that it holds, and then develop additional conditions on $\rightarrow_R$ that ensure full preservation of semantics. We need the following technical lemmata:

Lemma 1 *If $\rightarrow_D$ and $\rightarrow_N$ commute, then $R^* D^* t = R^* t$ for any term t.*

Proof. $R^* D^* t \subseteq R^* t$ is immediate, since $t \rightarrow_R^* D^* t$. Let us prove that $R^* D^* t \supseteq R^* t$, i.e. that if $t \rightarrow_R^! t'$, then $D^* t \rightarrow_R^! t'$. Since $\rightarrow_D$ is confluent and $\rightarrow_D$ and $\rightarrow_N$ commute, a simple diagram chase gives that also $\rightarrow_D$ and $\rightarrow_R$ commute. Therefore there must be a t'' such that $t' \rightarrow_D^* t''$ and $D^* t \rightarrow_R^* t''$. Since $\rightarrow_D^* \subseteq \rightarrow_R^*$ and t' is a normal form w.r.t. $\rightarrow_R$ we must have $t'' = t'$. $D^* t \rightarrow_R^! t'$ follows. ∎

Lemma 2 *If $\rightarrow_D$ and $\rightarrow_N$ commute, then, for any substitution σ and term t, $R^* \sigma D^* t = R^* \sigma t$.*

Proof. $t \rightarrow_D^* D^* t$ gives $\sigma t \rightarrow_D^* \sigma D^* t$, and subsequently $\sigma t \rightarrow_D^* D^* \sigma D^* t$. As in the proof of lemma 1, we directly have $R^* D^* \sigma D^* t \subseteq R^* \sigma t$. In the same way as in the proof of lemma 1 it also follows that also $R^* D^* \sigma D^* t \supseteq R^* \sigma t$. Finally, lemma 1 gives $R^* D^* \sigma D^* t = R^* \sigma D^* t$. ∎

Lemma 3 *For all substitutions σ and terms t holds that $R^* \sigma R^* t \subseteq R^* \sigma t$.*

Proof. All elements in $\sigma R^* t$ are of the form $\sigma t'$ where $t' \in R^* t$. Since $t \rightarrow_R^* t'$ it holds $\sigma t \rightarrow_R^* \sigma t'$, so there is a $t'' \in R^* \sigma t$ such that $\sigma t \rightarrow_R^* \sigma t' \rightarrow_R^! t''$. $R^* \sigma R^* t \subseteq R^* \sigma t$ follows. ∎

Theorem 3 *(Weak preservation of semantics of deterministic unfolding) If $\rightarrow_D$ and $\rightarrow_N$ commute, and if $\theta\pi = \pi\theta$, then $(R^*\pi)^*\theta(D^*\pi)^*t \supseteq (R^*\pi)^*\theta t$, whenever $(R^*\pi)^*$ is defined for e and θt.*

Proof. The proof is analogous to the proof of theorem 2 given in [13]. We prove by induction, for all $n \geq 0$, that

$$(R^*\pi)^*\theta(D^*\pi)^n t \supseteq (R^*\pi)^*\theta t.$$

- $n = 0$: obviously true.
- $n > 0$: assume true for $n-1$, i.e. $(R^*\pi)^*\theta(D^*\pi)^{n-1}t \supseteq (R^*\pi)^*\theta t$. We prove that $(R^*\pi)^*\theta(D^*\pi)^{n-1}t \subseteq (R^*\pi)^*\theta(D^*\pi)^n t$, which then yields the desired result:

$$\begin{array}{rcl}
(R^*\pi)^*\theta(D^*\pi)^{n-1}t & = & (R^*\pi)^*R^*\pi\theta(D^*\pi)^{n-1}t \\
& = & (R^*\pi)^*R^*\theta\pi(D^*\pi)^{n-1}t \\
& = & \text{(lemma 2)} \\
& = & (R^*\pi)^*R^*\theta D^*\pi(D^*\pi)^{n-1}t \\
& = & (R^*\pi)^*R^*\theta(D^*\pi)^n t \\
& = & (R^*\pi)^*R^*\pi R^*\theta(D^*\pi)^n t \\
& \subseteq & \text{(lemma 4)} \\
& \subseteq & (R^*\pi)^*R^*\pi\theta(D^*\pi)^n t \\
& = & (R^*\pi)^*\theta(D^*\pi)^n t.
\end{array}$$

■

A trivial modification of the proof above gives the following corollary:

Corollary 1 *(Preservation of semantics of deterministic unfolding) If $R^*\sigma R^*t = R^*\sigma t$, then $(R^*\pi)^*\theta(D^*\pi)^*t = (R^*\pi)^*\theta t$.*

Theorem 3 and corollary 1 can be trivially extended to sets of terms. As in the deterministic case [13] the proof above also shows that any "prematurely interrupted" unfolding (that is: $(D^*\pi)^n t$, where $(D^*\pi)^n t \neq (D^*\pi)^* t$) is also (weakly) semantics-preserving. We say that R is *correct* if $R^*\sigma R^*t = R^*\sigma t$ for all terms t and substitutions σ. There are indeed cases where we do not have full correctness, i.e. $R^*\sigma R^*t \subset R^*\sigma t$ where "$\subset$" is strict set inclusion. Let for instance R be given by

$$\begin{array}{rcl}
f(x) & \rightarrow_R & 1 \\
f(g(x)) & \rightarrow_R & 2,
\end{array}$$

let $t = f(x)$, and let $\sigma = \{x \leftarrow g(x)\}$. Then $R^*\sigma R^*t = R^*\sigma\{1\} = \{1\}$ and $R^*\sigma t = \{1, 2\}$.

A modification of this example illustrates why an unfolded program may produce *more* nondeterministic choices than the original program. Let now R be given by

$$\begin{array}{rcl}
f(x, 1) & \rightarrow_R & 1 \\
f(g(x), 1) & \rightarrow_R & 2 \\
f(g(f(x)), 1) & \rightarrow_R & 3,
\end{array}$$

let

$$\pi = \{h \leftarrow \lambda x.g(h(x))\}$$

and let $D = \beta$ (so the only deterministic simplification is β-reduction). With $t = f(h(x), y)$ and $\theta = \{y \leftarrow 1\}$ we have

$$R^*\pi\theta t = R^* f(g(h(x)), 1) = \{1, 2\}.$$

One deterministic unfolding, on the other hand, yields

$$D^*\pi t = f(g(h(x)), y).$$

Thus, $\theta(D^*\pi)t = f(g(h(x)), 1)$ and

$$R^*\pi\theta(D^*\pi)t = R^* f(g(f(h(x))), 1) = \{1, 2, 3\}.$$

Thus, the unfolded term can have more normal forms because successive applications of π can produce new redexes for nondeterministic reductions.

Incidentally, this example illustrates another possible pitfall when unfolding nondeterministic programs. $(D^*\pi)^*t$ is actually undefined, whereas $(R^*\pi)^*\theta t$ terminates in one step. This is because termination here is obtained through nondeterministic reductions. *Theorem 1 guarantees termination only relative to termination caused by deterministic reductions in D.*

We now consider criteria to ensure correctness of R. First, the deterministic case (cf. [13]):

Proposition 2 *If $\rightarrow_R$ is confluent (and terminating), then R is correct.*

Proof. Since then both $R^*\sigma R^* t$ and $R^*\sigma t$ are singletons, so they must be equal. ■

We will now prove correctness for a class of nonconfluent term rewriting systems. In the following, we restrict the scope to first order terms. Thus, β-reduction is not included in the rewrite rules. A slight modification of the evaluation model is then necessary: instead of modeling first order function definitions with (higher order) substitutions, they can be directly modeled by rewrite rules. For instance, a function definition expressed as

$$\{f \leftarrow \lambda x_1 \dots x_n.e\}$$

will be expressed as the rewrite rule

$$f(x_1 \dots x_n) \rightarrow e.$$

This is actually more in accordance with the tradition for recursive applicative program schemes: see, for instance [8] or [14]. All results so far carry over directly.

Definition 6 *A rewrite rule (s, t) is* flat *if s is of the form $f(x_1 \dots x_n)$, where $x_1 \dots x_n$ are variables.*

Examples of flat rules are the ones defining nondeterministic choice,

$$\begin{aligned} x + y &\rightarrow x \\ x + y &\rightarrow y, \end{aligned}$$

parallel if (a flat rule need not be left-linear),

$$\mathit{if}(x, y, y) \rightarrow y,$$

and any function definition $f(x_1 \dots x_n) = e$ can be seen as a flat rule of the form

$$f(x_1 \dots x_n) \rightarrow e.$$

The usual if-then-else-rules

$$\begin{array}{lcl} \mathit{if}(\mathbf{true}, x, y) & \rightarrow & x \\ \mathit{if}(\mathbf{false}, x, y) & \rightarrow & y, \end{array}$$

are, however, *not* flat since they contain the constants **true**, **false** in positions where a flat rule must have variables.

Theorem 4 *If R has only flat rules, then R is correct on first order terms.*

Proof. The proof follows by a kind of "parallel moves lemma". The important observation is that if a flat rule matches a position in $Pos(t)$ in a term σt, then the rule must also match the same position in t. Using this fact, we will prove that for every t' such that $\sigma t \rightarrow_R^* t'$ there is a v such that $t \rightarrow_R^! v$ and a v' such that $\sigma v \rightarrow_R^* v'$ and $t' \rightarrow_R^* v'$. It then follows that any normal form to σt is also a nomal form to some v in $\sigma R^* t$.

The proof is by induction over the length n of the derivation $\sigma t \rightarrow^n t'$. We will use a somewhat stronger induction hypothesis: that *all sequences of reductions from t', taking place in positions in $Res(Pos(t))$, will eventually result in a term v' such that $\sigma v \rightarrow_R^* v'$ for some normal form v to t.* We write t' $(p_1 \dots p_n)$ v' to state that a sequence of reductions, successively taking place in $p_1, \dots, p_n$, takes t' into v'. We also write $t'' \rightarrow_R^p t'$ if a reduction at p takes t'' into t'. (Not to be confused with $\rightarrow_R^n$.)

- $n = 0$: Obviously true, since $t \rightarrow_R^! v$ implies $\sigma t \rightarrow_R^* \sigma v$ where all redexes are in the original positions, i.e. in $Res(Pos(t))$.
- $n > 0$: assume true for $n - 1$. Then, if $\sigma t \rightarrow_R^n t'$, the induction hypothesis is true for any t'' such that $\sigma t \rightarrow_R^{n-1} t'' \rightarrow_R t$. Pick some. Then $t'' \rightarrow_R^p t'$ for some position p. There are two cases:
  1. $p \notin Res(Pos(t))$: Then, for any v'' and any sequence $(p_1 \dots p_n)$ of $Res(Pos(t))$-redexes such that t'' $(p_1 \dots p_n)$ v'', there are $m \geq 0$ disjoint positions in v'' that are residuals of p in t''. They will all be redexes w.r.t. the original reduction taking place at p. It follows that there is a term v' such that $v'' \rightarrow_R^m v'$ and t' $(p_1 \dots p_n)$ v'. Since the reduction at p did not affect any $Res(Pos(t))$-redex, the $Res(Pos(t))$-sequences from t' are the same as those from t''. (This is a "parallel move".)
  2. $p \in Res(Pos(t))$: then for any v'' and sequence $(pp_2 \dots p_n)$ of $Res(Pos(t))$-redexes such that t'' $(pp_2 \dots p_n)$ v'', we have t' $(p_2 \dots p_n)$ v''. Thus, all possible sequences $(p_2 \dots p_n)$ of $Res(Pos(t))$-reductions from t' must end in some v'' with $\sigma v \rightarrow_R^* v''$. (This is a "one step closer move".)

■

Corollary 2 *C, defined by*

$$\begin{array}{lcl} x + y & \rightarrow_C & x \\ x + y & \rightarrow_C & y, \end{array}$$

is correct. (Nondeterministic choice)

From the proof of theorem 4 it is easy to see why the restriction to first order terms is important. It is crucial for the correctness that if a flat rule matches a term σe, then it also matches e. But if higher order terms are allowed, then there

$$\begin{tikzcd} t'' \arrow[r, "R"] \arrow[d, "(p_1 \ldots p_n)"] & t' \arrow[d, "(p_1 \ldots p_n)"] \\ v'' \arrow[r, "R"] \arrow[r, "m" near end] & v' \end{tikzcd} \qquad \begin{tikzcd} t'' \arrow[dr, "R"] \arrow[dd, "(pp_2 \ldots p_n)"] & \\ & t' \arrow[dl, "(p_2 \ldots p_n)"] \\ v'' & \end{tikzcd}$$

Figure 2: A parallel move and a one step closer move.

are terms $t = x(t_1 \ldots t_n)$ where x is a free (higher order) variable. Consider a flat rule (s, s') with $s = f(x_1 \ldots x_n)$. (s, s') now matches $\{x \leftarrow f\}t = f(t_1 \ldots t_n)$ but not t.

Let us now turn to the the question of *compositionality*: if R_1 and R_2 are correct, when will $R_1 \cup R_2$ be correct? Apparently not always, as shown by the following simple counterexample. Let R_1 given by

$$f(x) \to_R 1$$

and let R_2 be given by

$$f(g(x)) \to_R 2.$$

Then both R_1 and R_2 are correct since each one is by itself confluent (left-linear and non-overlapping, see [12]). But as we saw before, $R_1 \cup R_2$ is not correct.

Lemma 4 *If for two term rewriting systems R_1 and R_2 holds that:*

- *R_1 and R_2 are left-linear and non-overlapping,*
- *for all rules $(s, s') \in R_1$ and $(q, q') \in R_2$, s' contains no function symbol in q, and*
- *R_2 is flat,*

then:

1. $\to^*_{R_1 \cup R_2} = \to^*_{R_2} \circ \to^*_{R_1}$.
2. *For any R_2-nf u, there is a $(R_1 \cup R_2)$-nf v such that $u \to^!_{R_1} v$.*

Proof. To prove 1, first consider three terms e, e', e'' where $e \to^*_{R_1} e' \to^*_{R_2} e''$. For the rules $(q, q') \in R_2$, q is of the form $f(x_1 \ldots x_n)$. No reduction in R_1 can introduce any occurrence of f: thus, any position p' of a R_2-redex in e' is a residual of a position p in e, where e/p is of the form $f(e_1 \ldots e_n)$. But then e/p is a R_2-redex. Since R_1 and R_2 are left-linear and non-overlapping, the parallel moves lemma can be applied: the origins in e of R_2-redexes in e' can be reduced first, followed by reductions of the residuals from e of R_1-redexes, to produce e''. It follows that $e \to^*_{R_2} e' \to^*_{R_1} e''$.

Now let $t \to^*_{R_1 \cup R_2} t'$. Then, there are terms $u_0, \ldots, u_n$ such that $t = u_0 \to^*_{R_1} u_1 \to^*_{R_2} \to^*_{R_1} u_3 \cdots \to^*_{R_2} u_n = t'$. A repeated application of the result above now yields $t \to^*_{R_2} \circ \to^*_{R_1} t'$.

To prove 2, consider any R_2-nf u. Consider any R_1-nf v such that $u \to^*_{R_2} v$. Then any R_2-redex in v must be a residual of some R_2-redex in u. But u is a R_2-nf: thus, v is also a R_2-nf and thus a $(R_1 \cup R_2)$-nf. ∎

$$
\begin{tikzcd}
\forall t \arrow[r, dotted, no head, "\sigma"] \arrow[d, "R_2"', "!"] & \sigma t \arrow[r, "R_2", "*"'] \arrow[d, "R_2", "*"'] & \exists t' \arrow[r, "R_1", "*"'] \arrow[d, "R_2", "*"'] & \forall t'' \arrow[d, "R_2", "*"'] \\
\exists u \arrow[r, dotted, no head, "\sigma"] \arrow[d, "R_1"', "!"] & \sigma u \arrow[r, "R_2", "*"'] \arrow[d, "R_1", "*"'] & \exists u' \arrow[r, "R_1", "*"'] \arrow[d, "R_1", "*"'] & \exists u'' \arrow[d, "R_1", "*"'] \\
\exists v \arrow[r, dotted, no head, "\sigma"] & \sigma v \arrow[r, "R_2", "*"'] & \exists v' \arrow[r, "R_1", "*"'] & \exists v''
\end{tikzcd}
$$

Figure 3: Illustration of the proof of theorem 5.

Theorem 5 *If for two term rewriting systems R_1 and R_2 holds that:*

- *R_1 is confluent,*
- *R_2 is correct,*
- *$\rightarrow_{R_1}$ and $\rightarrow_{R_2}$ commute,*
- *$\rightarrow^*_{R_1 \cup R_2} = \rightarrow^*_{R_2} \circ \rightarrow^*_{R_1}$, and*
- *For any R_2-nf u, there is a $(R_1 \cup R_2)$-nf v such that $u \rightarrow^!_{R_1} v$,*

then $R_1 \cup R_2$ is correct.

Proof. Consider some term t and any u where $t \rightarrow^!_{R_2} u$. Then there is a $(R_1 \cup R_2)$-nf v such that $u \rightarrow^!_{R_1} v$. We thus have $t \rightarrow^!_{R_1 \cup R_2} v$. We also have $\sigma t \rightarrow^*_{R_2} \sigma u \rightarrow^*_{R_1} \sigma v$. Consider now any t'' where $\sigma t \rightarrow^*_{R_1 \cup R_2} t''$. Then there is a t' such that $\sigma t \rightarrow^*_{R_2} t' \rightarrow^*_{R_1} t''$. Since R_2 is correct there is a u' such that $t' \rightarrow^*_{R_2} u'$ and $\sigma u \rightarrow^*_{R_2} u'$. We thus have $\sigma u \rightarrow^*_{R_2} u'$ and $\sigma u \rightarrow^*_{R_1} \sigma v$. $\rightarrow_{R_1}$ and $\rightarrow_{R_2}$ commute: thus, there is a v' such that $u' \rightarrow^*_{R_1} v'$ and $\sigma v \rightarrow^*_{R_2} v'$. Now, we have the situation that $t' \rightarrow^*_{R_2} u' \rightarrow^*_{R_1} v'$, i.e. $t' \rightarrow^*_{R_1 \cup R_2} v'$ and $t' \rightarrow^*_{R_1} t''$. Confluence of $\rightarrow_{R_1}$ and commutation with $\rightarrow_{R_2}$ yields commutation with $\rightarrow_{R_1 \cup R_2}$: thus, there is a v'' such that $t'' \rightarrow^*_{R_1 \cup R_2} v''$ and $\sigma v \rightarrow^*_{R_2} v' \rightarrow^*_{R_1} v''$ (or $\sigma v \rightarrow^*_{R_1 \cup R_2} v''$). The existence of such an element proves the correctness of $R_1 \cup R_2$. ■

Corollary 3 *If for two term rewriting systems R_1 and R_2 holds that:*

- *R_1 is confluent,*
- *R_2 is flat,*
- *R_1 and R_2 are left-linear and non-overlapping, and*
- *for all rules $(s, s') \in R_1$ and $(q, q') \in R_2$, s' contains no function symbol in q,*

then $R_1 \cup R_2$ is correct for first order terms.

Example: From the above we can deduce that any first order pure functional language, where no operators are defined by non-left-linear rules, can be extended with nondeterministic operations, defined by flat rules, in a way that makes it possible to perform semantics-preserving unfolding. Especially, nondeterministic choice can be added.

5 Conclusion and Further Research

We have extended results regarding unfolding of purely functional programs to programs with nondeterminism. The reduction system that gives semantics to the operators must be factored into a confluent and a possibly nonconfluent part, where the confluent part then gives semantics to the deterministic operators and the nonconfluent part gives semantics to the nondeterministic operators. Under certain conditions on the reduction system, correct unfolding can be done with regard to the confluent part. Termination results for unfolding of purely functional programs then carry over directly. It should be observed, however, that for programs whose termination properties depend on reduction of nonconfluent rules (execution of nondeterministic operators), unfolding where simplification is restricted to confluent rules cannot be expected to terminate, even when the termination criteria says so.

We pay certain attention to the issue of semantics-preservation of unfolding. The semantics of two nondeterministic programs are considered equivalent if the sets of possible results are equal. We restrict the scope to programs where all possible executions terminate. Under these conditions, we give some general conditions on the reduction system that ensures that unfolding preserves the semantics. There are two conditions: first, the confluent and nonconfluent part of the system must commute. Second, the whole reduction system must be *correct*, which essentially means that substitution cannot create any new nonconfluent redexes in a term through overlapping beteen the term and the substitution. The conditions are shown to be met by two kinds of systems: confluent systems in general, and *flat term rewriting systems* on first order terms. The latter can be nonconfluent: pure nondeterministic choice, for instance, can be defined by flat rules. We also give conditions for when a confluent term rewriting system and a flat term rewriting system can be combined into a system that is correct. These conditions essentially boil down to orthogonality and that the confluent part is not allowed to introduce any nondeterministic operators.

If the reduction system is not correct, then it seems like optimizing program transformations, if they are to preserve the semantics, must be restricted to parts of expressions that are known never to affect any nondeterministic choice of any kind. Therefore, establishing correctness is important from an implementation point of view.

The concept of semantics-preservation can be formulated for transformations on expressions in general. Thus, the possible preservation of semantics can be assessed for other source-to-source optimizing transformations than unfolding. If terms with explicit substitutions [1] are introduced, for instance, then transformations that yield sharing of subexpressions can be expressed as a term rewrite system with non-left-linear rules. E.g. the non-left-linear rule $f(x,x) \to f(y,y)[\![y \leftarrow x]\!]$ replaces common arguments to f with a single copy. (Here, $[\![y \leftarrow x]\!]$ is an explicit representation of a substitution where y is replaced by x. This should be considered a binding of y, so y is bound in $f(y,y)[\![y \leftarrow x]\!]$ and no free variables are thus introduced.) It then immediately follows that sharing of subexpressions can give incorrect results, since these rules will in general not commute with the nonconfluent rewrite rules.

We would like to point out the relationship with previous work on operational semantics for nondeterministic programs, [2, 5]. Proposition 6.6 in [2] is very much the same as our corollary 1 in the special case when a deterministic language is extended with pure nondeterministic choice. On the other hand, their result is with regard to a more developed semantics that takes infinite computations into account. Boudol [5] points out that in for computations defined by term rewriting systems in general, the full substitution evaluation rule is *not* correct, i.e. other evaluation orders can sometimes produce results that the full substitution evaluation rule cannot. This is related to our observation that symbolic unfolding of a program sometimes can increase the number of possible outcomes. Conversely, we conjecture that our "correctness" conditions on term reduction systems are sufficient to guarantee correctness (in the sense of Boudol) of the full substitution evaluation rule.

It would be practically very interesting to extend the results here to programs with processes. In order to do this several hurdles must be overcome, however. For instance, the operational semantics must be extended to deal both with infinite computations and with infinitely many possible outcomes. It is also not clear at all that simple reduction systems on terms as considered here will suffice to give semantics to a nontrivial class of process languages. Possibly other notions of rewriting (e.g. conditional, or modulo equivalence relations) can be useful and it would then be interesting to extend the results here to such systems. Again, this is a prime subject for future research.

6 Acknowledgements

This research was supported by the ESPRIT BRA project CONFER, project no. 6454.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Proc. Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, Jan. 1990.

[2] A. Arnold, P. Naudin, and M. Nivat. On semantics of nondeterministic recursive program schemes. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 1, pages 1–33. Cambridge University Press, 1985.

[3] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, 1981.

[4] G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. *J. Assoc. Comput. Mach.*, 26(1):148–175, Jan. 1979.

[5] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 5, pages 170–236. Cambridge University Press, 1985.

[6] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. Assoc. Comput. Mach.*, 24(1):44–67, Jan. 1977.

[7] J. M. Cadiou. *Recursive Definitions of Partial Functions and their Computation.* PhD thesis, Stanford University, Stanford, California, 1972.

[8] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9, pages 459–492. Elsevier Science Publishers B. V., 1990.

[9] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Inform.*, 6(1):41–60, 1976.

[10] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, Oct. 1980.

[11] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, Hertfordshire, UK, 1993.

[12] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, vol. 2*, chapter 1, pages 1–116. Oxford University Press, Oxford, 1992.

[13] B. Lisper. Total unfolding: Theory and applications. Accepted for publication in *Journal of Functional Programming*, 1993.

[14] J.-C. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *J. Assoc. Comput. Mach.*, 27(4):772–796, Oct. 1980.

[15] B. K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *J. Assoc. Comput. Mach.*, 20(1):160–187, Jan. 1973.