J. Functional Programming **4** (4): 479–498, October 1994

Total unfolding: theory and applications

BJÖRN LISPER

The Royal Institute of Technology, Department of Teleinformatics, Electrum/204, S-164 40 Kista, Sweden
and
Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden

Abstract

Unfolding is a common technique in program transformations. Here, we present a computation model where unfolding is a simple generalisation of the usual concept of evaluation. The model is a variant of the well-known full substitution evaluation rule for recursive programs. The evaluation mechanism involved is symbolic substitution of function definitions followed by simplification. Simplification is expressed as a confluent rewrite strategy which uses three kinds of reductions: β-reduction, non-erasing reduction, and erasing reduction. Non-erasing reductions include simplification of constant subexpressions. Erasing reductions formalize the behaviour of nonstrict operations. In this computation model, we prove a termination theorem of symbolic unfolding relative to more instantiated calls. A possible application of the model is a technique called *total unfolding*, where a partially instantiated function call is unfolded until no more calls exist. Under certain conditions the result will be a first order term: such terms correspond to basic blocks in imperative programs and can be efficiently implemented by scheduling techniques. Possible applications are hardware synthesis, and code generation for parallel machines.

Capsule Review

Symbolic evaluation of functional expressions with respect to a recursive program scheme is a key technique in *partial evaluation*, and if the result is a first order term it may be applied in hardware synthesis, and code generation for parallel machines. It is a difficult question, in general, when to stop evaluation, and whether it will lead to a first order term.

Björn Lisper formalizes the computation process as full substitution with three kinds of simplification: besides ordinary *β-reduction* he considers reductions defined by term-rewriting systems in which the rules are either *non-erasing* or *erasing*, where the latter may describe non-strict operations.

Using this model it is possible to prove that under certain conditions the result will be a first order term. Sufficient conditions for the termination of unfolding are also given, and the question of correctness of total unfolding is considered.

The paper presents an interesting combination of techniques from applicative program schemes, typed λ-calculus, and term rewriting systems.

1 Introduction

Operational semantics for applicative languages can be given by term rewriting systems. An example is *recursive applicative program schemes*, where recursive function definitions are considered as rewrite rules, possibly accompanied by additional rules that model 'basic operations' in the language. See Courcelle (1990) for a thorough introduction. Recursive applicative program schemes have been used to prove properties of different evaluation orders for applicative languages, like correctness (implements the least fixpoint solution) (Cadiou, 1972) and optimality (Berry & Lévy, 1979; Vuillemin, 1974). A salient feature of this kind of semantics is that it directly extends into symbolic computation: thus, it can be used to model various program transformations and prove properties of them. For instance, one can prove conditions for the correctness of the fold-unfold transformations pioneered by Burstall & Darlington (1977; 1976). (See Courcelle, 1990.)

A particular kind of program transformation is *unfolding*, where function definitions are substituted for function calls and formal arguments are replaced by actual (possibly symbolic) arguments. Unfolding is most often followed by a *simplification* phase where constant subexpressions are evaluated, branches are selected in conditionals with known conditions, etc. Unfolding and simplification are key techniques in *partial evaluation* (Bjørner *et al.* , 1988). In partial evaluation a general function, called with partially known arguments, is replaced by a specialized version, the 'residual', where the information about the arguments has been used to increase the efficiency of the code.

A problem with unfolding is how to know when to stop. Unfolding the wrong function calls may lead to code explosion, and the unfolding process itself may not terminate. Thus, it is important to have conditions when unfolding can proceed safely without risk of non-termination. Such conditions can be divided into two classes. The first class consists of conditions that really ensure termination. An example is Sestoft's *structural induction condition* (Sestoft, 1988). While these conditions yield safe results when succeeding, their applicability is limited: the reason is that the problem to be solved is the undecidable halting problem for symbolic computation. In practice one is restricted to a number of easily recognisable cases, unless theorem proving methods are used. The structural induction condition, for instance, requires that recursive function calls are made with arguments from the previous call 'staying in place', where each known argument is either unchanged or a substructure of the previous corresponding argument. This ensures that arguments always become strictly 'smaller' with respect to some non-negative measure which guarantees that only a finite number of calls can be made. The second class ensures 'weak termination', i.e. *under such a condition the unfolding will always terminate if a corresponding function call, with totally known arguments, terminates.* So this class of conditions relates termination of unfolding to termination of the function to be unfolded. While these conditions certainly are more 'unsafe', since one relies on the programmer to write terminating functions, they have the advantage that they can be put as simple syntactical tests. Furthermore, it turns out that the success of such tests often is correlated to other 'good' consequences of unfolding. These tests typically

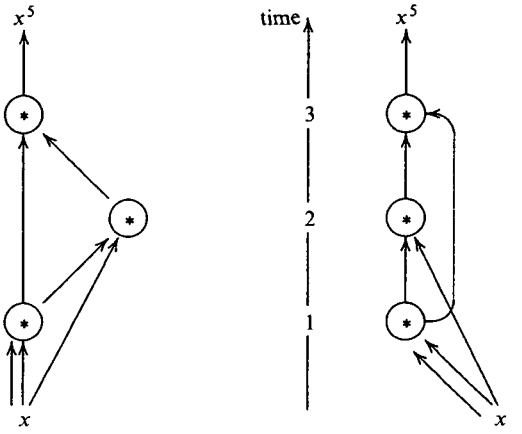

Fig. 1. Data dependence graph for computing x^5 and a uniprocessor schedule.

succeed when conditionals or the like can be evaluated and a branch be selected, which simplifies the control structure of the resulting code and often enables further simplifications. The partial evaluator Similix uses this kind of condition (Bondorf & Danvy, 1991).

In extreme cases, repeated unfolding can be done until termination. This means that there are no function calls left in the specialized function. Often the residual will be a first order term. Such terms correspond to *basic blocks* (Aho *et al.* , 1986) in imperative languages. Powerful code optimization techniques are applicable to first order terms, due to the fact that their data dependence graph is fully known. Such optimizations include the sharing of subexpressions, scheduling of operations and register allocation. As a simple example, consider the following function definition (similar to Dijkstra's 'obfuscation function' (1982). Also cf. Arsac & Kodratoff, 1882, sec. 4.3.4):

$$f(x,n) \leftarrow \mathit{if}(n = 1, x, f(x, \lceil n/2 \rceil) \cdot f(x, \lfloor n/2 \rfloor))$$

Obviously, this recursion equation describes a balanced algorithm for computing x^n when $n \geq 1$. Consider now the partially instantiated function call $f(x,5)$. A symbolic unfolding and simplification at each step yields:

$$\begin{aligned} f(x,5) &\leftarrow \mathit{if}(5 = 1, x, f(x, \lceil 5/2 \rceil) \cdot f(x, \lfloor 5/2 \rfloor)) \\ &\leftarrow f(x,3) \cdot f(x,2) \\ &\vdots \\ &\leftarrow (f(x,2) \cdot f(x,1)) \cdot (f(x,1) \cdot f(x,1)) \\ &\leftarrow ((f(x,1) \cdot f(x,1)) \cdot x) \cdot (x \cdot x) \\ &\leftarrow ((x \cdot x) \cdot x) \cdot (x \cdot x) \end{aligned}$$

The residual function is a first-order term that yields a 'static algorithm' (fixed dependence graph) for computing x^5. If the common subexpression $x \cdot x$ is shared, then the data dependence graph becomes that of Fig. 1, which also shows a possible

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

schedule on a uniprocessor. The following piece of imperative code will support the schedule:

```
tmp1   := x*x;
tmp2   := x*tmp1;
result := tmp1*tmp2
```

This code uses two registers and three multiplications, probably a substantial saving compared with a naïve applicative evaluation of $f(x, 5)$.

We may coin unfolding until termination 'total unfolding'. It has been argued that total unfolding is rarely possible except for trivial cases (Sestoft, 1988). This is, however, very dependent upon the application and the typical structure of its algorithms: *scientific computing* seems to be an area where total unfolding often is possible and beneficial (Berlin & Weise, 1990). The reason is that numerical algorithms often are controlled by simple 'size' parameters like, say, matrix dimensions: once these parameters are known, total unfolding will then succeed and first-order residuals will most often result. Note that such residuals are particularly amenable to implementation on fine-grain parallel and pipelined architectures. This is due to the predictability of the execution and the very simple control structure, which reduces control and synchronization overhead.

Algorithms given by first-order terms are especially suitable for implementation directly in hardware. Here, the scheduling can take place already at 'design-time'. Once a schedule is determined, a parallel system that supports the schedule with a minimum of control can be designed. In particular, regular hardware systems like *systolic arrays* (Kung, 1982) are amenable to synthesis by scheduling methods (Chen, 1986a; Delosme & Ipsen, 1987; Huang & Lengauer, 1987; Kung, 1987; Lisper, 1989; Lisper, 1990b; Moldovan, 1982; Quinton, 1984; Rajopadye & Fujimoto, 1990; Rao & Kailath, 1988). Also less regular processor structures than systolic arrays can be derived by scheduling of static algorithms, like for instance efficient VLSI structures for the Fast Fourier Transform algorithm (Lisper, 1989, 1990a).

In this paper, we investigate some properties of total unfolding. In order to do this, we develop an operational model of recursive symbolic computation where program transformations like unfolding can be readily expressed. The model is a higher order variant of recursive applicative program schemes and essentially formalizes the full substitution computation rule (Cadiou, 1972). This is a fixpoint computation rule: thus, our computation model is denotationally equivalent to lazy evaluation. Within the model we then show some theorems about total unfolding. Theorem 1 gives sufficient conditions for the unfolding to yield first order terms whenever it terminates. Theorem 3 states a general termination condition of the second class above, ensuring termination of the unfolding whenever the original function terminates. Syntactical tests can be based on this condition: some are given in theorem 4. These tests can guide unfolding either dynamically, as unfolding proceeds, or statically, in a preprocessing phase. For completeness, we also give a correcness result for unfolding in theorem 2.

2 Preliminaries

We assume that the reader is familiar with the basic concepts of λ-calculus (Barendregt, 1981; Hindley & Seldin, 1986). In this paper we consider terms in a λ-calculus with constants. We will sometimes consider a typed calculus where every well-formed term is assigned a simple type. In these type schemes, a type can be either a *base type* from a set B or a *function type* $\tau \to \tau'$ where τ, τ' are types. Terms of function type may be 'uncurried': then the type $\tau_1 \times \cdots \times \tau_n \to \tau$ is considered equivalent to $\tau_1 \to (\tau_2 \to \cdots (\tau_n \to \tau) \cdots)$. Abstraction and reduction will then sometimes be with respect to several arguments. We denote the set of free variables in the term e by $FV(e)$.

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. First order terms free from variables are called *ground 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 higher order term is *closed* if it has no free variables.

We will also consider a weaker form of typing where each constant is just assigned an *arity* ≥ 0. If f has arity n, then any term $f(t_1 \ldots t_m)$ where $m \leq n$ is well-formed. This makes it possible to express partially applied ('curried') functions.

We define a *substitution* to be a type-preserving partial function from variables to terms. We use set notation, where pairs $(x, \sigma(x)) \in \sigma$ are denoted $x \leftarrow \sigma(x)$. The substitution $\{(x, 17 + y), (y, y)\}$ is thus written as $\{x \leftarrow 17 + y, y \leftarrow y\}$. 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 $\sigma(x)$, i.e. $\bigcup(FV(\sigma(x)) \mid x \in dom(\sigma))$. A substitution is *finite* if its domain is finite. Substitutions can naturally be extended to general terms: $\sigma(e)$ is the term obtained when all occurrences in e of variables x in $dom(\sigma)$ are replaced with $\sigma(x)$. We distinguish between the original substitution (on variables) and the extended transformation on terms by writing $\sigma(x)$ when applying the former to a variable x, but σt (without parentheses) when applying the latter to a general term t. Substitutions on higher-order terms must include some renaming scheme to avoid capturing of free variables (see, for instance, Hindley & Seldin, 1986). *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.

For a binary relation '$\to$', we denote its transitive-reflexive closure with '$\to^*$'. The relation $\to$ is *terminating* (or *noetherian*) if there are no infinite sequences $a_0 \to a_1 \to \ldots \to a_i \to \ldots$. *Noetherian induction* can be done with respect to a terminating relation $\to$: to prove a predicate $P(x)$ for all x in the universe under consideration, one proves, for all x, that if $P(y)$ holds for all y such that $x \to y$, then $P(x)$ holds. This proof method will be frequently used here. See Huet (1980) for a more thorough introduction to noetherian relations and induction.

Throughout this paper we will, unless stated otherwise, use letters a, b, c for constants, e, r, s, t for general terms, f, g for functions or function symbols, Λ, p, u,

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

v, w for positions in terms (see next section), x, y, z for variables, and greek letters for transformations on terms and occasionally for types.

3 Reductions

Evaluation in general and partial evaluation in particular includes the notion of simplification. In this paper we will, besides the usual β-conversion, consider two kinds of simplification: *erasing* and *non-erasing* simplification. Non-erasing simplification means that no 'unknown' parts of a term are thrown away. An important kind of non-erasing simplification is δ-reduction which is defined for closed terms only. Erasing simplification means that parts of a term that possibly contain free variables are deleted. A typical example is simplification of a conditional, where one branch is taken and the other is discarded. Erasing simplification thus gives operational semantics to (predefined) function symbols whose denotational semantics are functions with nonstrict arguments, i.e. which may return defined values even when some certain arguments are undefined.

The two types of non-β-simplifications above will be modelled by *term rewriting systems*. For completeness, we give the main definitions involved, and some results that will be of use later. Details can be found in Huet (1980) and the survey by Klop (1992). The notation is a mix of Huet's and that suggested by Dershowitz & Jouannaud (1991).

First, we define some relevant concepts and state some properties for binary relations:

Definition 1
Let $\rightarrow$, $\rightarrow_1$, $\rightarrow_2$ be binary relations over some set A.

1. $a \in A$ is a *normal form* (w.r.t. $\rightarrow$) if there is no $b \in A$ such that $a \rightarrow b$.
2. $\rightarrow$ is *confluent* (or *Church-Rosser*) if $\forall a, b, c \in A \exists d \in A[(a \rightarrow^* b \wedge a \rightarrow^* c) \Longrightarrow (b \rightarrow^* d \wedge c \rightarrow^* d)]$.
3. $\rightarrow$ is *locally confluent* (or *weakly Church-Rosser*) if $\forall a, b, c \in A \exists d \in A[(a \rightarrow b \wedge a \rightarrow c) \Longrightarrow (b \rightarrow^* d \wedge c \rightarrow^* d)]$.
4. The equivalence relation generated by $\rightarrow$ is denoted $\overset{*}{\leftrightarrow}$. $\rightarrow$ has the *unique normal form property* if, for any normal forms a, b, $a \overset{*}{\leftrightarrow} b \Longrightarrow a = b$.

The following well-known result (Huet, 1980; Klop, 1992) is easily proved:

Proposition 1
A confluent relation has the unique normal form property.

A family of binary relations is sometimes called an *abstract reduction system*. Reduction systems model nondeterministic computations, since there may be more than one possible reduction for a given element. If the reduction system is confluent and terminating, however, then every element has a unique normal form which can seen as a deterministic 'function value' computed by the reduction system. We want the simplification phase of evaluation to always yield a well-defined result which means that we are mainly interested in confluent and terminating reduction systems.

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

Definition 2
Let $\rightarrow$ be a binary relation over A. A (one-step) *abstract reduction strategy F for* $\rightarrow$ is a function $F: A \rightarrow A$ such that:

1. $F(a) = a$ if a is a normal form.
2. $a \rightarrow F(a)$ otherwise.

A reduction strategy F models a 'deterministic implementation' of a reduction system $\rightarrow$. It is easy to see that F is terminating if $\rightarrow$ is terminating. Furthermore, if $\rightarrow$ is terminating and confluent, then F will always reach the normal form in a finite number of steps.

Definition 3
An abstract reduction strategy F for $\rightarrow$ is *confluent* if $\forall a, b, c \in A \exists n, m[(a \rightarrow^* b \wedge a \rightarrow^* c) \Longrightarrow F^n(b) = F^m(c)]$. F is *terminating for a* if there exists some $n \geq 0$ such that $F^n(a)$ is a normal form. We then write $F^*(a)$ for $F^n(a)$. F is *terminating* if it is terminating for all a under consideration.

(Cf. Barendregt, 1981, p. 351.) We have the following:

Proposition 2
If F is confluent and terminating, then $a \overset{*}{\leftrightarrow} b \iff F^*(a) = F^*(b)$.

Proof
Exactly analogous to the proof of lemma 2.1 in Huet (1980). □

Proposition 3
If F is terminating and if $\rightarrow$ is confluent, then F is confluent.

Proof
$F^*(x)$ exists for all x since F is terminating. If $a \rightarrow^* b$ and $a \rightarrow^* c$, then $F^*(b) = F^*(c)$ since these are normal forms w.r.t. $\rightarrow$ of a and normal forms, due to confluence of $\rightarrow$, are unique. Confluence of F follows. □

In particular, F is confluent and terminating whenever $\rightarrow$ is confluent and terminating. On the other hand, F can well be confluent and terminating even when $\rightarrow$ is not. So to make the theory more general we will actually model simplification by confluent and termination reduction strategies rather than reduction systems.

We now review the relevant definitions for reductions of terms. First 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 t we define its *set of positions* $Pos(t)$ and the *subterm at position* p, t/p, as:

- If $t = x$ or $t = c$, then $Pos(t) = \{\Lambda\}$ and $t/\Lambda = t$.
- If $t = \lambda x.t'$, then $Pos(t) = \{\Lambda\} \cup \{0p \mid p \in Pos(t')\}$, $t/\Lambda = t$ and $t/(0p) = t'/p$.
- If $t = t_0(t_1 \ldots t_n)$, then $Pos(t) = \{\Lambda\} \cup \{ip \mid 0 \leq i \leq n, p \in Pos(t_i)\}$, $t/\Lambda = t$ and $t/(ip) = t_i/p$.

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

Furthermore, we define $VPos(t) = \{ p \mid t/p \in FV(t) \}$ and $FPos(t) = Pos(t) \backslash VPos(t)$. We say that p is a *prefix* of p', or $p \leq p'$, if there exists a u such that $p' = pu$. u is then unique and we denote it with p'/p. If neither $p \leq p'$ nor $p' \leq p$, then p and p' are *disjoint*.

The *replacement in t with t' at p*, $t[p \leftarrow t']$, is defined by:

- $t[\Lambda \leftarrow t'] = t'$.
- $(\lambda x.t)[0p \leftarrow t'] = \lambda x.(t[p \leftarrow t'])$.
- $t_0(t_1 \ldots t_n)[ip \leftarrow t'] = t_0(t_1 \ldots t_i[p \leftarrow t'] \ldots t_n)$.

The following lemma is the direct generalisation of Proposition 3.4 in Huet (1980):

Lemma 1
If no variable in $dom(\sigma) \cup rg(\sigma)$ is bound in t, then it holds that:

1. $Pos(\sigma(t)) = Pos(t) \cup \bigcup_{p \in VPos(t)} \{ pp' \mid p' \in Pos(\sigma(t/p)) \}$.
2. For all $p \in Pos(t)$, $\sigma(t)/p = \sigma(t/p)$.

Definition 4
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 $(s,t) \in R \implies FV(s) \supseteq FV(t)$. 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 $FV(s) \supset FV(t)$, otherwise *non-erasing*.

One more technical definition is needed. To keep track of where subterms go when a term is rewritten, we must define the notion of *residual* (cf. Berry & Lévy, 1979). Formally, we express this as a relation between 'old' and 'new' positions. This relation is a function of the term e to be rewritten, the position p of the redex, and the matching rewrite rule (s,t).

Definition 5
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 $P \subseteq Pos(e)$, $Res(P,e,p,s,t) = \{ p'' \mid \exists p' \in P : p'\ Res(e,p,s,t)\ p'' \}$.

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 a term e' where $e \rightarrow_R^* e'$ (with the obvious definition for the case $e \rightarrow_R^0 e'$).

Let us now be a little more specific about the kinds of simplification we consider: First comes the usual β-conversion of applied terms: let β be the set of term pairs

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

$((\lambda x.t)(t'), \{x \leftarrow t'\}t)$ (with possible change of bound variable to avoid name clashes). We denote the finest relation containing β, closed under replacement, by $\rightarrow_\beta$. $\rightarrow_\beta$ is confluent, closed under substitution, and for certain λ-calculi, most notably the simply typed λ-calculus, it is also terminating. See, for instance, Thompson (1991).

Next comes non-erasing reductions: we model these by a term rewrite system v with only non-erasing rules. In particular, v can have *δ-rules* (t, t') where t and t' are closed terms. It is especially common to have δ-rules of the form (t, c), where c is a constant. δ-rules are intended to model evaluation of operators supported on 'machine-instruction level', like arithmetic and logical operations, e.g. $2 + 3 \rightarrow_v 5$, $true \vee false \rightarrow_v true$, etc.

Finally, we have erasing, or *nonstrict* reductions. We model these by a term rewriting system ϵ where a rule may be erasing. As an example, consider the following erasing rules, that give operational semantics to the usual *if*-function:

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

Adding the rule:

$$if(x, y, y) \rightarrow_\epsilon y$$

yields the 'parallel if'. Another example is the left-OR rule:

$$true \vee x \rightarrow_\epsilon true$$

which is common in logic programming languages. We will refer to rewritings with rules from ϵ as *nonstrict reductions.*

We denote $\rightarrow_v \cup \rightarrow_\epsilon$ with $\rightarrow_{\epsilon v}$ and $\rightarrow_v \cup \rightarrow_\beta \cup \rightarrow_\epsilon$ with $\rightarrow_{\beta\epsilon v}$. This relation is closed under replacement and substitution since it is a union of such relations. Of particular interest are term rewriting systems v, ϵ where $\rightarrow_{\beta\epsilon v}$ is also confluent and terminating, since then every reduction strategy must be confluent and terminating as well. A result of Dougherty (1992) is particularly useful in this context and we give it below. First a definition; for a term rewriting system R, a set of terms is *R-stable* iff:

1. It is closed under taking subterms and under $\rightarrow_{\beta R}$ (i.e. $\rightarrow_\beta \cup \rightarrow_R$),
2. Each term in the set is strongly β-normalizing, and contains no subterm of the form $f(e_1 \ldots e_n)$ where n is greater than the arity of f.

The main result of Dougherty is now the following:

- If R is confluent, then $\rightarrow_{\beta R}$ is confluent on R-stable terms.
- If R is terminating, then $\rightarrow_{\beta R}$ is terminating on R-stable terms.

Thus, under the weak condition of R-stability, the question whether $\rightarrow_{\beta\epsilon v}$ is confluent and terminating reduces to the question whether $\rightarrow_{\epsilon v}$ is confluent and terminating. How to prove such properties of term rewriting systems is a well researched subject. Again, we refer to the survey by Klop (1992) for a comprehensive discussion of proof techniques.

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

4 Recursive programs

In this section we give a formalization of recursive programs. Essentially it is a kind of recursive applicative program scheme (Courcelle, 1990), albeit put in a form somewhat more convenient for our purposes. The operators of the language under consideration have their semantics defined by term rewriting systems ν and ϵ as defined in section 3.

Definition 6
A *recursive program* π is a finite substitution where $rg(\pi) \subseteq dom(\pi)$, no variable in $dom(\pi)$ occurs in any rule of ν or 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. Reconsider the example in section 1. The definition of the balanced x^n-algorithm can be formally put as the substitution:

$$\{f \leftarrow \lambda xn.\, if(n = 1, x, f(x, \lceil n/2 \rceil) \cdot f(x, \lfloor n/2 \rfloor))\}$$

The *full substitution evaluation rule* is known to be a fixpoint computation rule (Cadiou, 1972), 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).

Now, let the simplification phase (β-, ν-, ϵ-reduction) be defined by a confluent and terminating reduction strategy F for $\rightarrow_{\beta\epsilon\nu}$. This strategy defines a total function F^* on the terms under consideration, that maps a term to its normal form. One full step of the full substitution rule can then 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 7
Let t be a term. Then, if for some $n > 0$ it holds that $(F^*\pi)^n t$ is free from variables in $dom(\pi)$, we define $(F^*\pi)^* t = (F^*\pi)^n t$. The evaluation of t *terminates* if $(F^*\pi)^* t$ is defined.

5 Properties of total unfolding

In section 4 we formalized evaluation by the full substitution rule as a partial transformation $(F^*\pi)^*$ of terms. Applied to function calls with fully instantiated arguments it always yields a ground term when terminating. In this section we study the case where $(F^*\pi)^*$ is applied to general terms. When the term is a function call with partially instantiated arguments, $(F^*\pi)^*$ can be seen as a simple partial evaluator.

Reconsider now the x^n-example in section 1. It is immediately clear that the

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

step by step-derivation of the first order term for x^5 really is a tracing of the transformations of $(F^*\pi)^*$. Thus:

$$(F^*\pi)^* f(x, 5) = ((x \cdot x) \cdot x) \cdot (x \cdot x)$$

When $F^*\pi$ is applied to a term e, all occurrences in e of variables defined by π are replaced by their definitions. $(F^*\pi)^*$ performs what can be called *total unfolding*; it will either terminate, with all occurrences of defined functions unfolded, or not terminate at all. The following theorem gives a sufficient condition under which total unfolding, when terminating, yields a first order term (i.e. a static algorithm). We assume a typing scheme with base types and function types.

Theorem 1

Assume that for any rule $(s, t) \in \nu \cup \epsilon$, any higher order constant function symbol in t also occurs in s. Let π be a recursive program without higher order constant function symbols. Let e be a term of base type whithout higher order constant function symbols and where any free higher order variable is defined by π. Then $(F^*\pi)^* e$, when defined, is a first order term.

Proof

Assume that $(F^*\pi)^* e$ is defined and that the condition in the theorem holds. In a first order term the only subterms with a function type are first order function symbols. Terms of function type, except for these, are higher order variables, higher order (constant) function symbols and λ-abstractions. We will prove that no such subterms can occur in $(F^*\pi)^* e$ under the given conditions:

- *Higher order variables*: since neither F nor π introduce any new variables, and since $(F^*\pi)^* e$ is free of variables in $dom(\pi)$, there can be no free higher order variables in $(F^*\pi)^* e$. Any bound higher order variables must occur in λ-abstractions, see below.
- *Higher order constant function symbols*: there are no higher order constant function symbols in e. π cannot introduce any new. In F, the potential places for introduction of new such symbols is reduction of first order terms $e_0(e_1 \dots e_n)$, by ν- or ϵ-reduction $\sigma s \rightarrow_{\epsilon\nu} \sigma t$, where $(s, t) \in \nu \cup \epsilon$ and $\sigma s = e_0(e_1 \dots e_n)$. From the assumption on $\nu \cup \epsilon$, the only possible occurrence of a higher order function symbol in σt is in $\sigma(x)$, for some variable $x \in FV(s)$. But $\sigma(x)$ is then a subterm of the original term, so no new function symbols are introduced. By induction it now follows that $(F^*\pi)^* e$ is free from higher order constant function symbols.
- *λ-abstractions*: $(F^*\pi)^* e$ has no β-redex, so it contains no applied λ-abstractions. Furthermore, e is of base type, and so is $(F^*\pi)^* e$ since $(F^*\pi)^*$ obviously does not change the type. Thus, any possible λ-abstraction in $(F^*\pi)^* e$ must occur as an argument e_i in an term $e_0(e_1 \dots e_n)$. e_0 is then of higher order type. By the above, no such e_0 can occur in $(F^*\pi)^* e$. □

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

Corollary 1
Under the conditions in theorem 1, any free variable in $(F^*\pi)^*e$ is a first order variable occurring in e.

Higher order constant function symbols typically appear in programs as representations for 'system-defined' higher order functions. An example is the the function symbol '*apply*'. At a first glance, the conditions of theorem 1 seem to disqualify programs with such functions. It is, however, often possible to add formal definitions of such functions to π. This turns the constants into defined functions, that eventually are substituted away whenever $(F^*\pi)^*$ terminates. For instance, a definition $apply \leftarrow \lambda f x.f(x)$ may be added to π, in which case $(F^*\pi)^* apply(\lambda x.x+x, y) = y+y$.

Correctness of unfolding can be formulated as follows. Consider a term t and a substitution θ such that $dom(\theta) \cap dom(\pi) = \emptyset$. θt is then an *instance* of t, where free variables x possibly have been replaced by $\theta(x)$. So we may choose to apply the original functions in π to θt, i.e. calculate $(F^*\pi)^*\theta t$. As an alternative, we may choose the unfolded version $(F^*\pi)^*t$ (if defined), instantiate the free variables, and finally evaluate the result. This means that we compute $(F^*\pi)^*\theta(F^*\pi)^*t$. If unfolding is correct, these two terms should be equal. We will now prove that this is indeed the case. For simplicity, we assume that $rg(\theta)$ is free from functions defined by π: since $dom(\theta) \cap dom(\pi) = dom(\theta) \cap rg(\pi) = \emptyset$, this means that θ and π are totally disjoint, and thus $\theta\pi = \pi\theta$.

Lemma 2
For any term t and substitution σ, $F^*\sigma t = F^*\sigma F^*t$.

Proof
$t \rightarrow_{\beta\epsilon v} F^*t$ implies $\sigma t \rightarrow_{\beta\epsilon v} \sigma F^*t$. Proposition 2 now gives the result. □

An immediate consequence of lemma 2 is that $(F^*\pi)^n t = F^*\pi^n t$ for all $n > 0$. Proposition 11 of Raoult & Vuillemin (1980) says essentially the same thing, but in a somewhat more restricted computation model.

Theorem 2
(Correctness of unfolding) $(F^*\pi)^*\theta(F^*\pi)^*t = (F^*\pi)^*\theta t$, whenever $(F^*\pi)^*$ is defined for the terms in question.

Proof
We prove by induction, that it for all $n \geq 0$ holds that $(F^*\pi)^*\theta(F^*\pi)^n t = (F^*\pi)^*\theta t$:

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

$$\begin{aligned}
(F^*\pi)^*\theta(F^*\pi)^{n-1}t &= (F^*\pi)^*F^*\pi\theta(F^*\pi)^{n-1}t \\
&= (F^*\pi)^*F^*\theta\pi(F^*\pi)^{n-1}t \\
&= \text{(lemma 2)} \\
&= (F^*\pi)^*F^*\theta F^*\pi(F^*\pi)^{n-1}t \\
&= (F^*\pi)^*F^*\theta(F^*\pi)^n t
\end{aligned}$$

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

$$
\begin{aligned}
&= (F^*\pi)^* F^*\pi F^*\theta (F^*\pi)^n t \\
&= \text{(lemma 2, again)} \\
&= (F^*\pi)^* F^*\pi\theta (F^*\pi)^n t \\
&= (F^*\pi)^*\theta (F^*\pi)^n t. \qquad \square
\end{aligned}
$$

As can be seen from the proof, any 'prematurely interrupted' unfolding (that is: $(F^*\pi)^n t$, where $(F^*\pi)^n t \neq (F^*\pi)^* t$) is also correct (cf. Courcelle, 1990, where a similar result is given).

We now consider termination. It is essential to have a criterion that decides when a function call can be unfolded without risk of nontermination. Theorem 3 gives such a criterion, where termination of the unfolding of a partially instantiated function call is related to the termination of a call with totally instantiated arguments. It is based on a family of relations '$\geq_X$', for sets of variables X, between terms. $t \geq_X t'$ means roughly that t is less instantiated than t' but they agree on the variables in X.

Definition 8
For any set of variables X, the relation $\geq_X$ between terms t, t' is defined by: $t \geq_X t'$ iff there is a substitution θ such that $dom(\theta) \cap X = \emptyset$ and $\theta t \rightarrow_v^* t'$.

Some examples: if $x \notin X$ then $x + 3 \geq_X 2 + 3$, and $x \geq_X 5$ since $2 + 3 \rightarrow_v 5$. But if $x \in X$, then $x + 3 \not\geq_X 2 + 3$ and $x \not\geq_X 5$, and in no case it holds that $\lambda x.(x + 3) \geq_X \lambda x.(2 + 3)$. $\geq_X$ also compares structured data, like lists, if we consider them as terms built with the constructors for the datatype. Thus, $a::(x::NIL) \geq_X a::(b::NIL)$ whenever $x \notin X$, and $L \geq_X a::(x::NIL)$ whenever $L \notin X$. Before proving the theorem we now state and prove some technical lemmata.

Lemma 3
If $t \geq_X t'$ and $X' \subseteq X$, then $FV(t) \cap X' \subseteq FV(t') \cap X'$.

Proof
$\theta t \rightarrow_v^* t' \implies FV(\theta t) = FV(t')$ since $\rightarrow_v$ is non-erasing. Thus, the result follows if $FV(t) \cap X' \subseteq FV(\theta t) \cap X'$ whenever $dom(\theta) \cap X = \emptyset$. Since no variable in X' may be substituted away from t by θ, the result follows directly. $\square$

Lemma 4
For any substitution σ, if $t \geq_{X \cup dom(\sigma)} t'$ and if $rg(\sigma) \subseteq dom(\sigma)$, then $\sigma t \geq_X \sigma t'$.

Proof
There is a θ, where $dom(\theta) \cap (X \cup dom(\sigma)) = \emptyset$, such that $\theta t \rightarrow_v^* t'$. Thus, $\sigma\theta t \rightarrow_v^* \sigma t'$. Now, by the definition of composition of substitutions, $\sigma\theta = \{ x \leftarrow \sigma(\theta(x)) \mid x \in dom(\theta) \} \cup \sigma|_{dom(\sigma) \setminus dom(\theta)} =$ (since $dom(\sigma) \cap dom(\theta) = \emptyset$) $= \{ x \leftarrow \sigma(\theta(x)) \mid x \in dom(\theta) \setminus dom(\sigma) \} \cup \{ x \leftarrow \sigma(x) \mid x \in dom(\sigma) \}$. Define θ' by $dom(\theta') = dom(\theta)$ and $\theta'(x) = \sigma(\theta(x))$ for $x \in dom(\theta)$. For $x \in dom(\sigma)$ it then holds that $\theta'(x) = x$. Since $rg(\sigma) \subseteq dom(\sigma)$ we have $\sigma(x) = \theta'(\sigma(x))$ for $x \in dom(\sigma)$. It follows that $\sigma\theta = \{ x \leftarrow \theta'(x) \mid x \in dom(\theta') \setminus dom(\sigma) \} \cup \{ x \leftarrow \theta'(\sigma(x)) \mid x \in dom(\sigma) \} = \theta'\sigma$. Thus, $\theta'\sigma t \rightarrow_v^* \sigma t'$, i.e. $\sigma t \geq_X \sigma t'$. $\square$

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

The following definition formalizes a situation where a term in a 'critical' position may prevent a nonstrict rewrite rule to match, while a more instantiated term would possibly match. To avoid this situation is essential for termination of unfolding.

Definition 9
t has a *potential* ϵX*-match* iff there exists a term t' such that $t \geq_X t'$ and t'/p is an ϵ-redex for some $p \in Res(FPos(t))$. t has a *potential applied* νX*-match* iff there exists a term t' such that $t \geq_X t'$, for some $p \in Res(FPos(t))$, where t'/p is a ν-redex that is applied, i.e. occurs as $(t'/p)(t'')$. t has a *potential* βX*-match* iff it has a subterm of the form $x(t')$, where $x \notin X$. t has a *potential* X*-match* iff it has a potential ϵX-, βX-, or νX-match.

Thus, a term has a potential ϵX-match if there is a way to instantiate the variables outside X and simplify the resulting term so the result matches a nonstrict reduction in a position occurring in the original term. Similarly, a potential applied νX-match exists if there is a way to instantiate the variables outside X and simplify so a higher order ν-reduction can take place and the result be applied to an argument. Finally, a potential βX-match exists if there is a way to instantiate a variable outside X so a β-redex is created. (For instance, $x(t) \geq_X (\lambda y.17)(t)$ when $x \notin X$.) The next four lemmata provide the major part of the forthcoming result.

Lemma 5
Let $dom\,\theta \cap X = \emptyset$. If t is a normal form w.r.t. $\rightarrow_{\beta\epsilon\nu}$ that has no potential X-match, and if $\theta t \rightarrow_\nu^* t'$, then t' has no β- or ϵ-redex in $Res(FPos(t))$.

Proof
Apparently, t' can have no ϵ-redex or applied ν-redex in $Res(FPos(t))$. We prove by induction over $\rightarrow_\nu^n$ that no β-redex ever appears:

- $\theta t \rightarrow_\nu^0 t'$: t is a normal form: thus, it has no β-redex. Since there is no potential βX-match, θ cannot introduce any β-redex in $FPos(t) = Res(FPos(t))$.
- $\theta t \rightarrow_\nu^{n+1} t'$: Assume the statement is true for t'' where $\theta t \rightarrow_\nu^n t'' \rightarrow_\nu t'$. Since t'' has no β-redex in $Res(FPos(t))$, the only possible way of introducing a β-redex in $Res(FPos(t))$ of t'' is through a ν- or ϵ-reduction $t'/p \rightarrow_\nu \lambda x.t''$, $p \in Res(FPos(t))$, where t'/p occurs in a context $(t'/p)(t''')$. But since there are no potential applied νX- or ϵX-matches, this cannot occur. □

In the next lemma, $\{F^*\theta\}$ denotes the substitution $\{F^*\theta(x) \mid x \in dom(\theta)\}$. (Note that $\{F^*\theta\}t \neq F^*(\theta t)$ in general.)

Lemma 6
If F^*t has no potential X-match then, for each substitution θ where $dom(\theta) \cap X = \emptyset$, it holds that $\{F^*\theta\}(F^*t) \rightarrow_\nu^* F^*(\theta t)$.

Proof
By noetherian induction over the terminating subrelation of $\rightarrow_{\beta\epsilon\nu}$ defined by F:

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

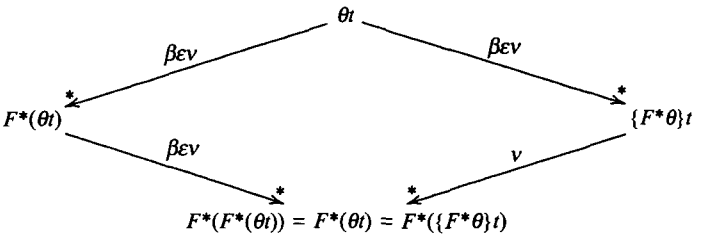

Fig. 2. Illustration of the base case of the proof of lemma 6.

- Base case, $t = Ft$. Thus, $F^*t = t$. Consider θt. We have $\theta t \to^*_{\beta\varepsilon\nu} \{F^*\theta\}t$ and $\theta t \to^*_{\beta\varepsilon\nu} F^*(\theta t)$. Confluence of F gives $F^*(\{F^*\theta\}t) = F^*(F^*(\theta t)) = F^*(\theta t)$ which implies $\{F^*\theta\}t \to^*_{\beta\varepsilon\nu} F^*(\theta t)$. There can be no redexes of any sort in any $F^*\theta(x)$: thus, any reductions affecting $\{F^*\theta\}t$ and subsequent terms must occur at positions in $Res(FPos(t))$. By lemma 5 then follows that $\{F^*\theta\}t \to^*_\nu F^*(\theta t)$. An illustration of the situation is given in figure 2.
- Inductive case, $t \to_{\beta\varepsilon\nu} Ft$: assume the theorem is true for Ft. $F^*Ft = F^*t$, so if F^*t has no potential X-match of any of the kinds above, then $\{F^*\theta\}(F^*t) \to^*_\nu F^*(\theta(Ft))$. Furthermore, $t \to_{\beta\varepsilon\nu} Ft$ and substitutivity of $\to_{\beta\varepsilon\nu}$ yields $\theta t \to_{\beta\varepsilon\nu} \theta(Ft)$. Proposition 2 then gives $F^*(\theta t) = F^*(\theta(Ft))$. $\{F^*\theta\}(F^*t) \to^*_\nu F^*(\theta t)$ follows. □

Lemma 7

If $t \geq_X t'$ and F^*t does not have any potential X-match, then $F^*t \geq_X F^*t'$.

Proof

$t \geq_X t'$ means that $\theta t \to^*_\nu t'$ for some θ. Thus, by proposition 2, $F^*(\theta t) = F^*t'$. Lemma 6 then yields $\{F^*\theta\}(F^*t) \to^*_\nu F^*t'$, i.e. $F^*t \geq_X F^*t'$. □

Theorem 3

(Relative termination of unfolding) If $dom(\pi) \subseteq X$, and if it for all $n > 0$ holds that $(F^*\pi)^n t$ has no potential X-match, then $t \geq_X t'$ implies that $(F^*\pi)^*t$ is defined whenever $(F^*\pi)^*t'$ is defined.

Proof

If $t \geq_X t'$ and if $dom(\pi) \subseteq X$, then lemma 4 is applicable and $\pi t \geq_X \pi t'$ follows. By assumption, $(F^*\pi)t = F^*(\pi t)$ has no potential X-match. Lemma 7 then yields $F^*\pi t \geq_X F^*\pi t'$. Simple induction over n now gives $(F^*\pi)^n t \geq_X (F^*\pi)^n t'$ also for all $n > 1$. Since $dom(\pi) \subseteq X$, lemma 3 gives that $FV((F^*\pi)^n t) \cap dom(\pi) \subseteq FV((F^*\pi)^n t') \cap dom(\pi)$ for all $n \geq 0$. Thus, $(F^*\pi)^n t$ is free from variables in $dom(\pi)$ whenever $(F^*\pi)^n t'$ is. □

Consider the classical pattern for partial evaluation: a partially instantiated function call $f(b, x)$, where b is static (constant) and x is dynamic (i.e. unknown, a variable), relative to a totally instantiated function call $f(b, c)$ where both b and c are static. Whenever f is defined by π, where $dom(\pi) \subseteq X$ and $x \notin X$, it holds that

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

$f(b,x) \geq_X f(b,c)$. Theorem 3 can then be applied. The theorem is, however, not limited to this simple instance of partial evaluation: it can also, for instance, be applied to situations where a structured argument (say, a list) is partially instantiated.

When theorem 3 is used to certify termination of unfolding, it must be checked for each application of $F^*\pi$ whether the result has a potential X-match or not. A direct application of definition 9 to decide this is not feasible, however: that would require applying each possible substitution with domain distinct from X and check whether there is any way to v-reduce the result so that any subterm matches a nonstrict reduction. Fortunately, there are often simple conditions that are sufficient to prevent any potential X-matches. Below, $LPos(t)$ denotes the set of 'leaf' positions in the term t, i.e. the positions in $Pos(t)$ that are not prefixes to any other positions in $Pos(t)$.

Theorem 4

Assume that v contains only δ-rules of the form (t,c), where c is a constant. Then e has no potential ϵX-match if, for each position $p \in FPos(e)$, some of the following holds ($e_p = e/p$ below): for any $(s,t) \in \epsilon$:

1. There is a $p' \in Pos(s)$ that has no prefix in $VPos(e_p)$.
2. There is a $p' \in Pos(e_p) \setminus Pos(s)$ such that there is no prefix to p' in $LPos(s)$.
3. There is a $p' \in Pos(s)$ where s/p' is a closed term, e_p/p' is a normal form w.r.t. $\rightarrow_{\beta\epsilon v}$, $e_p/p' \neq s/p'$, and $FV(e_p/p') \subseteq X$.
4. There are distinct positions $p', p'' \in Pos(s)$ where $s/p' = s/p''$, e_p/p' and e_p/p'' are normal forms w.r.t. $\rightarrow_{\beta\epsilon v}$, $e_p/p' \neq e_p/p''$, $FV(e_p/p') \subseteq X$ and $FV(e_p/p'') \subseteq X$.

Proof

We will prove for any e' such that $e \geq_X e'$ that if $u \in Pos(e')$ is a residual of p, then e'/u cannot be a ϵ-redex, i.e. there is no σ such that $\sigma s = e'/u$ for any $(s,t) \in \epsilon$. So assume there is a θ with $dom(\theta) \cap X = \emptyset$ such that $\theta e \rightarrow_v^* e'$. Since v-rules are of the form (t,c), either p has no residual in e' (a reduction took place at a prefix of p) or $\theta e_p = (\theta e)/p \rightarrow_v^* e'/u$:

1. By lemma 1, there is no σ such that $\sigma s = \theta e_p$, i.e. θe_p cannot be a ϵ-redex. Since v-rules of the form (t,c) cannot match any subterm with free variables it holds that $VPos(e'/u) = VPos(e_p)$: thus, e'/u cannot be a ϵ-redex either.
2. Since $VPos(s) \subseteq LPos(s)$ lemma 1 yields that there is no σ such that $\sigma s = \theta e_p$. By the same reason the only way for s to match e'/u is if all positions in $Pos(e'/u) \setminus Pos(s)$ have a prefix in $LPos(s)$. This can only happen if, for some e'' where $e_p \rightarrow_v^* e'' \rightarrow_v^* e'/u$, some v-rule matches e'' at a position in $Pos(s) \setminus LPos(s)$ that is a prefix to p'. But then $(e'/u)/p' = c$ which gives $p' \in LPos(e'/u)$. By lemma 1 follows that there is no σ such that $p' \in LPos(\sigma s)$, so $LPos(e/u) \neq LPos(\sigma s)$ and e'/u cannot be a ϵ-redex.
3. We have $\theta(e_p/p') = e_p/p'$ since $dom(\theta) \cap X = \emptyset$. Since e_p/p' is a normal form, $(e'/u)/p' = e_p/p'$. Furthermore, $(\sigma s)/p' = s/p'$ since s/p' is closed. $(e'/u)/p' \neq (\sigma s)/p'$ follows. Again, e'/u cannot be a ϵ-redex.

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

4. Similarly, it follows that $(e'/u)/p' \neq (e'/u)/p''$. Since it must hold that $(\sigma s)/p' = (\sigma s)/p''$, e'/u cannot be a ϵ-redex. □

The requirements on ν can be weakened in some cases: in case one it is for instance sufficient that $t \rightarrow_\nu t' \implies Pos(t) \supseteq Pos(t')$, which relaxes the condition that ν only has δ-rules (t, c). The three first conditions can also be used to rule out the existence of potential applied νX-matches. Potential βX-matches, finally, can always be found directly. Note also that there are quite common situations when νX- and βX-matches are not interesting. If there are no higher order ν-rules, then there is no need to check for potential applied νX-matches. All results proved here will still hold. Furthermore, in a first order language where the only higher order variables are functions defined by π, there cannot be any potential β-matches.

Some examples: let $\epsilon = \{if(true, x, y) \rightarrow x, if(false, x, y) \rightarrow y\}$. Furthermore, we assume that $f \in X$ and that $x, y \notin X$:

- $if(f(17), x, y)$ is, by theorem 4, not a potential ϵX-match (so any unfolding can safely proceed). Any term that is 'more instantiated' must still have $f(17)$ in a position where the left-hand side of both rewrite rules has a constant, which prevents the term from reducing non-strictly.
- $if(x, x, y)$ is a potential ϵX-match.
- $f(\mathbf{true}, x, y)$ is not a potential ϵX-match.
- $x(\mathbf{true}, x, y)$ is a potential ϵX-match. x, here a higher order variable, may be instantiated to *if*. It is also a potential βX-match and a potential applied νX-match.

Let us now consider $\epsilon' = \epsilon \cup \{if(y, x, x) \rightarrow x\}$. Then, the following holds:

- $if(f(17), x, y)$ is a potential $\epsilon' X$-match. x and y may well be instantiated to the same value.
- $if(f(17), x, x + 1)$ is a potential $\epsilon' X$-match. Actually it cannot reduce, but to infer this would require the 'semantical knowledge' that $x \neq x + 1$ always.
- $if(f(17), 7, 8)$ is not a potential $\epsilon' X$-match. Here, the 'syntactic knowledge' that $7 \neq 8$ is used to decide that a reduction $if(y, x, x) \rightarrow x$ cannot take place.

Theorems 1, 3 and 4 provide a basis for unfolding recursive programs to static terms. A practical method based on theorem 3 can operate in two ways. It can prove syntactically, from the structure of the program, that certain arguments can be safely left uninstantiated without causing non-termination. The other way is to actually perform the symbolic evaluation. Then the method must have some mechanism, for instance based on theorem 4, to decide when the condition in the theorem is fulfilled and then stop the unfolding. This may result in partially unfolded programs, where the method has found parts of the programs which are static.

Theorem 3 is based on a computational approach where the value of a term becomes known only by evaluating it to the extent that it is known whether or not it will match any nonstrict reduction. A partial evaluator may, however, be able to decide the value of a term in a 'non-constructive' way, without evaluating it at all,

https://doi.org/10.1017/S0956796800001155 Published online by Cambridge University Press

e.g. $e = e$, $b \vee \neg b$ and $x :: NIL \neq NIL$ can all be found to be *true* without evaluating e, b, and x, respectively. This can be seen as allowing the rewrite strategy F to use additional rewrite rules, which are not present at runtime. Such rules will not have to be checked for potential X-matches, since they will not be used at runtime. If the rewrite strategy is still terminating, then these rules can only improve the termination properties from pure unfolding techniques based on theorem 3.

6 Conclusions

We gave a formalization of the full substitution computation rule, where simplification of terms is expressed as a confluent and terminating rewrite strategy using ν-, β- and ϵ-reductions, where the latter may be erasing and thus give operational semantics to nonstrict operations. Through this formalization, the computation rule is directly extended to symbolic evaluation of general terms. This leads to a form of partial evaluation which in its pure form is called total unfolding. When total unfolding succeeds a static algorithm, often in the form of a first order term, remains. We gave a theorem connecting termination of total unfolding with termination of the corresponding evaluation when the function arguments are fully instantiated. Under certain conditions, the termination properties are the same. These conditions can be checked during the unfolding to stop if it is found not to be safe.

It is not always necessary to perform an explicit unfolding to find that total unfolding will succeed. In the x^n-example in section 1, for instance, it is easy to see that the only variable ever affecting the conditional is n. It follows that for any value of $n \geq 0$, $f(x, n)$ will be unfolded to a finite first-order term. Apparently, techniques like *Known/Unknown abstract interpretation* (Sestoft, 1985), normally used for *binding-time analysis*, can be useful. Methods to schedule dependence graphs directly from function definitions, without actually creating the graphs, have a potentially great practical importance, since in practice the unfolded graphs can be prohibitively large. Another consequence is that if a scheduling strategy is found that is *parameterised* in the static argument (n in the x^n-example), then an imperative implementation is found for the *general* function, with *all* arguments dynamic. An area where this is actually done is synthesis of regular hardware structures from simple classes of recurrences (Chen, 1986b; Karp *et al.*, 1967; Quinton, 1984; Rajopadye & Fujimoto, 1990; Rao & Kailath, 1988; Shang & Fortes, 1989). Investigating and generalising such techniques is a very interesting thread for further research.

It seems like the greatest potential for techniques based on total unfolding lies in the areas where functions with relatively simple structure are used. Typical such areas are scientific computing and signal processing. These areas also require highly efficient code: conventionally executed functional programs are as a rule totally inadequate for these purposes. Numerical algorithms are often defined by straightforward recurrences that in many cases should be possible to treat by total unfolding techniques. Recent work regarding partial evaluation in scientific computing show encouraging results (Berlin & Weise, 1990). Total unfolding and scheduling may provide new compilation methods for applicative languages that

make them more suitable to scientific computing. In fact, when it comes to parallelism and vectorization, such methods should have a greater potential for success than parallelizing compilation of imperative languages, since the former only have to deal with pure data dependencies while the latter are hampered by the artefacts of the inherently seqential semantics for imperative languages.

7 Acknowledgements

I would like to thank Karl-Filip Faxén for interesting discussions, Peter Sestoft for providing me with a recent version of his annotated bibliography on partial evaluation and mixed computation, and the anonymous referees for valuable suggestions and comments that helped improve the manuscript a great deal. Many thanks also to Jana Eriksson for her kind assistance with practical matters. This research was partially supported by The Swedish Research Council for Engineering Sciences under grant 91-333, and in part by the ESPRIT BRA project CONFER, project no. 6454.

References

Aho, A. V., Sethi, R. & Ullman, J. D. (1986) *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

Arsac, J. & Kodratoff, Y. (1982) Some techniques for recursion removal from recursive functions. *ACM TOPLAS* **4**(2), 295–322.

Barendregt, H. P. (1981) *The lambda calculus – its syntax and semantics: Studies in Logic and the Foundations of Mathematics, vol. 103.* North-Holland.

Berlin, A. & Weise, D. (1990) Compiling scientific code using partial evaluation. *Computer* **23**(12), 25–37.

Berry, G. & Lévy, J.-J. (1979) Minimal and optimal computations of recursive programs. *J. ACM.* **26**(1), 148–175.

Bjørner, D., Ershov, A. P. & Jones, N.D. (eds) (1988) *Partial evaluation and mixed computation: Proc. IFIP TC2 Workshop*, Gammel Avernæs, Denmark, October. North-Holland.

Bondorf, A. & Danvy, O. (1991) Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Program.* **16**, 151–195.

Burstall, R. M. & Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM* **24**(1), 44–67.

Cadiou, J. M. (1972) *Recursive definitions of partial functions and their computation.* PhD thesis, Stanford University, CA.

Chen, M. C. (1986a) A design methodology for synthesizing parallel algorithms and architectures. *J. Parallel Distrib. Comput.* 461–491.

Chen, M. C. (1986b) A parallel language and its compilation to multiprocessor machines or VLSI. In: *Proc. Principles of Programming Languages*, pp. 131–139, January.

Courcelle, B. (1990) Recursive applicative program schemes. In: *Handbook of Theoretical Computer Science*, J. van Leeuwen (ed.), Chapter 9, pp. 459–492. Elsevier.

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

Delosme, J.-M. & Ipsen, I. (1987) Efficient systolic arrays for the solution of Toeplitz systems: an illustration of a methodology for the construction of systolic architectures in VLSI. In: *Systolic Arrays*, W. Moore, A. McCabe & R. Urquhart (eds.), pp.37–46. Adam Hilger.

Dershowitz, N. & Jouannaud, J.-P. (1991) Notations for rewriting. *Bull. Euro. Assoc. Theoretical Computer Science*, 162–172, February.

Dijkstra, E. W. (1982) An exercise for Dr. R. M. Burstall. In: *Selected writings – A personal perspective on computer science*, pp. 215–216. Springer-Verlag.

Dougherty, D. J. (1992) Adding algebraic rewriting to the untyped lambda calculus. *Information and Computation* **101**, 251–267.

Hindley, J. R. & Seldin, J. P. (1986) *Introduction to Combinators and λ-calculus: London Mathematical Society Student Texts, vol. 1.* Cambridge University Press.

Huang, C.-H. & Lengauer, C. (1987) The derivation of systolic implementations of programs. *Acta Inform.* **24**, 595–632.

Huet, G. (1980) Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM* **27**(4), 797–821.

Karp, R. M., Miller, R. E. & Winograd, S. (1967) The organization of computations for uniform recurrence equations. *J. ACM* **14**(3), 563–590.

Klop, J. W. (1992) Term rewriting systems. In: *Handbook of Logic in Computer Science, vol.2*, Abramsky, S., Gabbay, D. M. & Maibaum, T. S. E. (eds.), Chapter 1, pp. 1–116. Oxford University Press.

Kung, H. T. (1982) Why systolic architectures? *Computer* **15**, 37–46.

Kung, S. Y. (1987) VLSI array processors. In: *Systolic Arrays*, Moore, W., McCabe, A. & Urquhart, R. (eds.), pp. 7–24. Adam Hilger.

Lisper, B. (1989) *Synthesis of synchronous systems by static scheduling in space-time: Lecture Notes in Computer Science, vol. 362.* Springer-Verlag.

Lisper, B. (1990a) The Interactive Space-Time Scheduler. *Microprocess. Microprogram.* **30**(1–5), 109–116.

Lisper, B. (1990b) Synthesis of time-optimal systolic arrays with cells with inner structure. *J. Parallel Distrib. Comput.* **10**(2), 182–187.

Moldovan, D. I. (1982) On the analysis and synthesis of VLSI algorithms. *IEEE Trans. Comput.* **31**, 1121–1126.

Quinton, P. (1984) Automatic synthesis of systolic arrays from uniform recurrent equations. In: *Proc. 11th Ann. Int. Symp. on Comput. Arch.*, pp. 208–214

Rajopadye, S. V. & Fujimoto, R. M. (1990) Synthesizing systolic arrays from recurrence equations. *Parallel Computing* **14**, 163–189.

Rao, S. K. & Kailath, T. (1988) Regular iterative algorithms and their implementation on processor arrays. *Proc. IEEE* **76**(3), 259–269.

Raoult, J.-C. & Vuillemin, J. (1980) Operational and semantic equivalence between recursive programs. *J. ACM* **27**(4), 772–796.

Sestoft, P. (1985) The structure of a self-applicable partial evaluator. In: *Programs as Data Objects: Lecture Notes in Computer Science, Vol. 217*, Ganzinger, H. & Jones, N. P. (eds.). Springer-Verlag.

Sestoft, P. (1988) Automatic call unfolding in a partial evaluator. In: *Partial Evaluation and Mixed Computation: Proc. IFIP TC2 Workshop*, Gammel Avernæs, Denmark, pp. 485–506. North-Holland.

Shang, W. & Fortes, J. A. B. (1989) On the optimality of linear schedules. *J. VLSI Signal Process.* **1**, 209–220.

Thompson, S. (1991) *Type Theory and Functional Programming.* Addison-Wesley.

Vuillemin, J. (1974) Correct and optimal implementations of recursion in a simple programming language. *J. Computer and System Sci.* **9**, 332–254.