A New Means of Ensuring Termination of Deforestation With an Application to Logic Programming

Morten Heine Sørensen*

Abstract

Wadler's *deforestation* algorithm eliminates intermediate data structures from functional programs, but is only guaranteed to terminate for a certain class of programs. Chin has shown how one can apply deforestation to all first-order programs. We develop a new technique of ensuring termination of deforestation for all first-order programs which strictly extends Chin's technique in a sense we make precise.

We also show how suitable modifications may render our technique applicable to ensure termination of transformers of logic programs such as *partial evaluation* as studied by Gallagher and others and the *elimination procedure* studied by Proietti and Pettorossi.

1 Introduction

Modern functional programming languages such as Miranda[1] [Tur90] lend themselves to a certain elegant style of programming which exploits *higher-order functions*, *lazy evaluation* and *intermediate data structures*; Hughes [Hug90] gives illuminating examples.

While this programming style makes it easy to read and write programs, it also results in inefficient programs. Consequently, researchers are struggling to find techniques to transform an (elegant) inefficient program into a semantically equivalent (inelegant) efficient program. This paper is concerned with a transformation which eliminates intermediate data structures.

*Authors address: DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark. E-mail:rambo@diku.dk+

[1]Miranda is a trademark of Research Software Ltd.

EXAMPLE **1** Consider the Miranda-like program that computes the sum of the first n natural numbers:

$$\begin{array}{lcll} ss\ n & \leftarrow & sum\ (upto\ 1\ n) & \\ sum\ Nil & \leftarrow & 0 & \\ sum\ (Cons\ x\ xs) & \leftarrow & x + sum\ xs & \\ upto\ m\ n & \leftarrow & Nil & \text{if } m > n \\ & \leftarrow & Cons\ m\ (upto\ (m+1)\ n) & \text{otherwise} \end{array}$$

The function $ss\ n$ first constructs an intermediate list of the integers from 1 to n and then sums the elements. This allocation and deallocation of cons-cells at run-time is expensive. Sacrificing clarity for efficiency, we would prefer a program like the following.

$$\begin{array}{lcll} h\ 0\ 1\ n & & & \\ \text{where} & & & \\ h\ a\ m\ n & \leftarrow & a & \text{if } m > n \\ & \leftarrow & h\ (a+m)\ (m+1)\ n & \text{otherwise} \end{array}$$

The inefficiency has been removed since no intermediate list is constructed. □

Wadler invented a technique to transform functional programs of the former kind into the latter, *deforestation* [Wad88, Fer88], and proved that the deforestation algorithm terminated when applied to so-called *treeless* programs.

Later a fully automatic technique for applying deforestation to arbitrary first-order terms was described by Chin [Chi90, Chi93]. He invented an *extended deforestation* algorithm which essentially leaves *annotated* subterms untransformed. The problem remains to calculate *safe* annotations, *i.e.* annotations ensuring that application of the extended deforestation algorithm to the annotated program terminates. Obviously as few subterms as possible should be annotated and hence left untransformed. Chin's syntactic technique essentially annotates non-treeless subterms.[2]

This paper describes a new static analysis whose result can be used to ensure termination of deforestation in such a way that fewer annotations will be put on the program, compared to Chin's technique.

[2]By Chin's technique we mean a certain, simplified version of Chin's actual method, the former described in [Sor93a]. While the subsequent comparisons with Chin's method should be read in the context of this reservation, on-going correspondence with Chin does seem to suggest that the simplified version is a reasonable core of his actual method.

The deforestation algorithm has also been invented in the Logic Programming community by Proietti and Pettorossi as the *elimination procedure* [Pro90, Pro91]. Their technique for ensuring termination is very similar to Chin's for deforestation. This suggests that it may be possible to find a technique extending Proietti and Pettorossi's similar to ours. Further, deforestation of functional programs is very similar to partial evaluation of logic programs as described in *e.g.* [Gal93]; indeed, in our view more similar than partial evaluation of functional and logic programs. This suggests that it may be possible to apply our technique for deforestation to partial evaluation of logic programs.

This paper takes the first steps in applying our technique to ensure termination of the elimination procedure and partial evaluation of logic programs.

2 The deforestation algorithm

We study the same language as the one studied in [Fer88].

DEFINITION 1 (Language.)

$$
\begin{array}{lcll}
t & ::= & v \mid c\,t_1 \ldots t_n \mid f\,t_1 \ldots t_n \mid g\,t_0 \ldots t_n & \text{(variables, constructors, function calls)} \\
p & ::= & c\,v_1 \ldots v_n & \text{(patterns)} \\
d & ::= & f\,v_1 \ldots v_n \leftarrow t & (f\text{-function definition}) \\
 & \mid & g\,p_1\,v_1 \ldots v_n \leftarrow t_1 & \\
 & & \vdots & (g\text{-function definition with patterns}) \\
 & & g\,p_m\,v_1 \ldots v_n \leftarrow t_m &
\end{array}
$$

We use h to range over functions which are either f- or g-functions.

As is customary, we require that patterns be *linear*, *i.e.* that no variable occur more than once. We also require that all variables of a right hand side of a definition be present in the corresponding left hand side. To ensure uniqueness of reduction, we require from a program that each function have at most one definition and, in the case of a g-definition, that no two patterns p_i and p_j contain the same constructor.

The semantics for reduction of a variable-free term is the usual *lazy evaluation*, like in Miranda. □

Apart from the fact that the language is first-order there are two obvious restrictions on the language. Firstly, function definitions may have at most

one pattern matching argument and only with non-nested patterns. This restriction is not essential; it is adopted simply to be able to state the deforestation algorithm concisely. Also, methods exist for translating arbitrary patterns into the restricted form [Aug85, Wad87].

Secondly, patterns are linear. One can extend deforestation to non-linear patterns or, equivalently, to a language which can test equality on terms. In doing so, certain interesting problems do appear, see [Sor93b].

Before we can state the deforestation algorithm we introduce some notation.

In deforestation, as in a rewrite interpreter, we need some notation that allows us to pick a function call in a term and replace the call by the body of the function with arguments substituted for formal parameters. Since the deforestation algorithm basically simulates lazy evaluation, there is always a unique function call whose unfolding is *forced.* For instance, to find out which of g's clauses to apply to $g\ (f_1\ t_1)\ (f_2\ t_2)$ we are forced to unfold the call $f_1\ t_1$. In the terminology below, the forced call $f_1\ t_1$ is the *redex* and the surrounding term $g\ []\ (f_2\ t_2)$ is the *context.*

DEFINITION **2** (Context, redex and observable.) Let e, r, o range over *(lazy evaluation) contexts*, *redexes* and *observables*, respectively, as defined by the grammar:

$$\begin{array}{lcl} e & ::= & [] \mid g\ e\ t_1 \ldots t_n \\ r & ::= & f\ t_1 \ldots t_n \mid g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n \mid g\ v\ t_1 \ldots t_n \\ o & ::= & c\ t_1 \ldots t_n \mid v \end{array}$$

The expression $e[t]$ denotes the result of replacing the occurrence of $[]$ in e by t. □

Note that every term t is either an observable or decomposes uniquely into a context e and redex r such that $t \equiv e[r]$ (*the unique decomposition property.*) This provides the desired way of "grabbing hold" of the next function call to unfold in a term.

The deforestation algorithm we shall consider works on annotated programs. The idea is that if a function is dangerous to unfold we associate $\ominus$ with it, and if it is dangerous to substitute t for v when unfolding a call to h with t as argument for v then we put v in h's set of dangerous arguments (h's *index set.*)

DEFINITION **3** (Annotation.) Given a program p, the following is an annotation a of p. Every definition defines a function f or g. With a f-function

definition with formal parameters $x_1 \ldots x_n$, a associates one $\oplus$ or $\ominus$ and a set $I \subseteq \{x_1 \ldots x_n\}$. With a g-function definition with patterns $p_1 \ldots p_m$ and formal parameters (apart from those in patterns) $v_1 \ldots v_n$, a associates one $\oplus$ or $\ominus$, a set $I \subseteq \{v_1 \ldots v_n\}$, and m sets $I_{c_1} \ldots I_{c_m}$, where I_{c_i} is a subset of the set of variables occurring in p_i. The sets I, I_{c_k} are called *index* sets.

We say that a is the trivial annotatation if a associates $\oplus$ with every function and all index sets are empty.

Given two annotations a_1, a_2 of p we write $a_1 \sqsubseteq a_2$ (a_1 *is better than* a_2) if a_2 associates $\ominus$ with at least all the function definitions that a_1 associates $\ominus$ with, and every index set in a_2 contains at least all the parameters of the corresponding set in a_1.

Given two annotations a_1, a_2 of p we define a new annotation $a_1 \sqcup a_2$ of p as follows. If either a_1 or a_2 associates $\ominus$ with h, $a_1 \sqcup a_2$ associates $\ominus$ with h as well, and otherwise $\oplus$. Every index set in $a_1 \sqcup a_2$ is the union of the two corresponding index sets from a_1 and a_2. $\Box$

Note that worse annotations lead to less transformation. The trivial annotation is the best of all annotations, but it will often be the case that deforestation on a trivially annotated program does not terminate. For a given program, the set of annotations on it is partially ordered by $\sqsubseteq$, and $\sqcup$ is the least upper bound (*lub.*)

The following definition is essentially the extended deforestation algorithm in [Chi93].

DEFINITION 4 (Generalizing folding deforestation algorithm)

$$
\begin{array}{lll}
(1) & \mathcal{D}[\![\, v \,]\!]ds & \Rightarrow \quad v \\
(2) & \mathcal{D}[\![\, c\, t_1 \ldots t_n \,]\!]ds & \Rightarrow \quad c\ (\mathcal{G}[\![\, t_1 \,]\!]ds) \ldots (\mathcal{G}[\![\, t_n \,]\!]ds) \\
(3) & \mathcal{D}[\![\, e[f\, t_1 \ldots t_n] \,]\!]ds & \Rightarrow \quad f'\, u_1 \ldots u_k \\
 & \quad \text{where} & \\
 & \quad f'\, u_1 \ldots u_k & \leftarrow \quad \mathcal{G}[\![\, e[\, t^f[v_i^f := t_i]_{i=1}^n \,] \,]\!]ds \\
(4) & \mathcal{D}[\![\, e[g\ (c\, t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n] \,]\!]ds & \Rightarrow \quad f'\, u_1 \ldots u_k \\
 & \quad \text{where} & \\
 & \quad f'\, u_1 \ldots u_k & \leftarrow \quad \mathcal{G}[\![\, e[\, t^{g,c}[v_i^{g,c} := t_i]_{i=1}^{n+m} \,] \,]\!]ds \\
(5) & \mathcal{D}[\![\, e[g\, v\, t_1 \ldots t_n] \,]\!] & \Rightarrow \quad g'\, v\, u_1 \ldots u_k \\
 & \quad \text{where} & \\
 & \quad g'\, p_1^g\, u_1 \ldots u_k & \leftarrow \quad \mathcal{G}[\![\, e[t^{g,c_1}[v_i^{g,c_1} := t_i]_{i=1}^n] \,]\!]ds \\
 & & \quad \vdots \\
 & \quad g'\, p_m^g\, u_1 \ldots u_k & \leftarrow \quad \mathcal{G}[\![\, e[t^{g,c_m}[v_i^{g,c_m} := t_i]_{i=1}^n] \,]\!]ds
\end{array}
$$

$$
\begin{array}{lllll}
(1) & \mathcal{F}[\![\ o\ ]\!]ds & \Rightarrow & \mathcal{D}[\![\ o\ ]\!]ds & \\
(2) & \mathcal{F}[\![\ e[r]\ ]\!]ds & \Rightarrow & h\ u_1 \ldots u_k, & \text{if } h\ u_1 \ldots u_k \leftarrow e[r] \in ds \\
 & & \Rightarrow & \mathcal{D}[\![\ e[r]\ ]\!](ds \cup \{h\ u_1 \ldots u_k \leftarrow e[r]\}), & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
(1) & \mathcal{G}[\![\ o\ ]\!]ds & \Rightarrow \mathcal{F}[\![\ o\ ]\!]ds \\
(2) & \mathcal{G}[\![\ e[h^I_{\ominus}\ t_1 \ldots t_n]\ ]\!]ds & \Rightarrow \mathrm{let}_{v_i^h \in I}\ v_i = \mathcal{G}[\![\ t_i\ ]\!]ds, v = \mathcal{F}[\![\ h\ u_1 \ldots u_n[u_i := v_i, u_j := t_j]_{v_i^h \in I, v_j^h \notin I}\ \\
 & & \quad \text{in } \mathcal{G}[\![\ e[v]\ ]\!]ds \\
(3) & \mathcal{G}[\![\ e[h^I_{\oplus}\ t_1 \ldots t_n]\ ]\!]ds & \Rightarrow \mathrm{let}_{v_i^h \in I}\ v_i = \mathcal{G}[\![\ t_i\ ]\!]ds \\
 & & \quad \text{in } \mathcal{F}[\![\ e[h\ u_1 \ldots u_n[u_i := v_i, u_j := t_j]_{v_i^h \in I, v_j^h \notin I}]\ ]\!]ds
\end{array}
$$

□

This calls for a number of explanations.

Notation

The algorithm should be understood in the context of some program p.

For f-functions, t^f denotes the right hand side of the definition for function f in p, and $v_1^f \ldots v_n^f$ are the formal parameters in f's definition.

For g-functions, $t^{g,c}$ is the right hand side of g corresponding to the left hand side whose pattern contains the constructor c. Further, $v_1^{g,c} \ldots v_n^{g,c}, v_{n+1}^{g,c} \ldots v_{n+m}^{g,c}$ are the formal parameters of g in the clause whose pattern contains the constructor c. Here $v_1^{g,c} \ldots v_n^{g,c}$ are those not occurring in the pattern (these are the same for all c) while $v_n^{g,c}, \ldots v_{n+m}^{g,c}$ are the vairables in the pattern.

The expression $t[v_i := t_i]_{i=1}^n$ denotes the result of simultaneously replacing all occurrences of v_i in t by t_i for all $i = 1 \ldots n$.

Function $\mathcal{D}$

In clauses (3-5) the term is transformed into a call to a new function[3] whose right hand side (before further transformation) is defined to be the original term unfolded one step. This is an *unfold* step and a *define* step. In clause (5) an *instantiation* step is performed as well.

We can make the significance of the unfold steps more apparent by using a slightly different notation for the argument term in clauses (3-5):

[3] We can imagine that the where-construct is present in our language, or that these new functions are collected somehow in a new program. We take the liberty of being imprecise on this point.

$$
\begin{array}{ll}
g_1\ (g_2\ \ldots(g_n\ [f\ t_1 \ldots t_k]\ t_1^n \ldots t_{m_n}^n) \ldots)\ t_1^1 \ldots t_{m_1}^1 & (n \geq 0) \\
g_1\ (g_2\ \ldots[g_n\ (c\ t_1 \ldots t_k)\ t_1^n \ldots t_{m_n}^n] \ldots)\ t_1^1 \ldots t_{m_1}^1 & (n > 0) \\
g_1\ (g_2\ \ldots[g_n\ v\ t_1^n \ldots t_{m_n}^n] \ldots)\ t_1^1 \ldots t_{m_1}^1 & (n \geq 0)
\end{array}
$$

where the terms $t_1^i \ldots t_{m_i}^i$ are arguments of g_i. Using the terminology of Chin [Chi90] (and his later works), the idea[4] here is that the redex through a number of unfoldings evaluates to a term with an outermost constructor, *produces a constructor*. This will allow the surrounding g-function to be unfolded, *consuming* exactly the outermost constructor from the term, since patterns are one constructor deep. This latter unfolding will itself through a number of subsequent unfoldings produce a constructor allowing the next surrounding g-function to be unfolded. In this way, the constructor propagates all the way to the root of the term, and transformation then proceeds to each of the arguments of the constructor in a similar fashion.

Function $\mathcal{F}$

The set ds contains definitions where the right hand sides are the terms that $\mathcal{F}$ has previously seen and the left hand sides are invented function names and parameter lists containing all the free variables of the corresponding right hand sides. Specifically, the formal parameters $u_1 \ldots u_k$ invented in clause (2) in $\mathcal{F}$ are calculated as follows. If the redex has form $g\ v\ t_1 \ldots t_n$, then $u_1 \ldots u_k$ are v followed by all the variables of $e, t_1 \ldots t_n$ (including v if present.[5]) Otherwise, $u_1 \ldots u_k$ are simply all the variables of $e[r]$.[6]

Before passing its term to $\mathcal{D}$, $\mathcal{F}$ checks whether the term has previously been seen. (The membership test is to be carried out modulo variable renaming.) If so, it performs a *fold* step. Otherwise, it passes its argument to $\mathcal{D}$ and recalls the new term. This ensures that none of $\mathcal{D}, \mathcal{F}, \mathcal{G}$ encounters the same term an infinite number of times. (However we shall see that they may encounter infinitely many different terms.) The function name and parameter list invented in $\mathcal{D}$ for the new function f' or g' is chosen as the left hand side of the definition which has just been added to ds by $\mathcal{F}$.

It is worth noting that $\mathcal{F}$ introduces a new function definition and call in *every* step in which it does not fold. This function is redundant if its right

[4] The following description represents the *desired* situation; the algorithm does not behave this well on all programs, as we shall see.

[5] This is actually a very important point; see [Sor93b] and section 8.

[6] Actually, one has do a renaming of the term and then chose the formal parameters as above; we ignore the details.

hand side in ds is never again encountered. We assume that such redundant calls are eliminated by a post processing phase.

Function $\mathcal{G}$

The let-construct is not present in our language, but we can consider is an abbreviation of $f\,t$ where f is defined as $f\,v \Leftarrow t'$ and v does not occur in t. A let with several assignments let $v_1 = t_1, \ldots, v_n = t_n$ in t' can be translated into nested let's.

Now, before passing control to $\mathcal{F}$, $\mathcal{G}$ performs an *extraction* step: it extracts the entire redex if the function is annotated $\ominus$, and it extracts arguments corresponding to parameters in index sets. More precisely, faced with a call $e[g\,v\,t_1 \ldots t_n]$ or $e[f\,t_1 \ldots t_n]$, $\mathcal{G}$ will extract those terms among $t_1 \ldots t_n$ which have their corresponding formal parameter in I. Faced with a call $g\,(c_k\,t_{n+1} \ldots t_{n+m})\,t_1 \ldots t_n$, algorithm $\mathcal{G}$ will extract those terms among $t_1 \ldots t_{n+m}$ which have their corresponding formal parameter in $I \cup I_{c_k}$.

Note in particular that $\mathcal{G}$ performs code generation.

Unlike redundant function calls introduced by $\mathcal{F}$, we cannot uncritically unfold the let's (function calls) introduced by $\mathcal{G}$ in a post processing phase without risking a serious degradation of the resulting program's efficiency due to duplication of computation, see [Sor93a] chapter 5. We assume that there is a post processing phase which unfold let's whenever there is no risk of degradation of efficiency.

To deforest a term t in a program p we annotate the program and apply $\mathcal{G}$ to the term (in the context of the program.) Applying $\mathcal{G}$ to a trivially annotated program is the same as applying Wadler's deforestation algorithm to the unannotated program (Wadler's deforestation algorithm is essentially $\mathcal{F}$ if one replaces the calls to $\mathcal{G}$ in $\mathcal{D}$ with calls to $\mathcal{F}$.)

There are some well-known problems concerning the efficiency of the program which deforestation outputs, due to possible duplication of computation. These problems are solved in the usual way, see [Sor93a] Chapter 5.

3 Termination problems in deforestation

Below we give two example programs which show the two kinds of problems that can occur. We show that application of $\mathcal{G}$ to each of them with the trivial annotation loops infinitely. We also show that with certain non-trivial annotations $\mathcal{G}$ does not loop infinitely.

EXAMPLE **2** (The Accumulating Parameter.)

$$\begin{array}{lcl} & & r\ l \\ r\ xs & \leftarrow & rr\ xs\ Nil \\ rr\ Nil\ ys & \leftarrow & ys \\ rr\ (Cons\ z\ zs)\ ys & \leftarrow & rr\ zs\ (Cons\ z\ ys) \end{array}$$

The r function returns its argument list reversed. With the trivial annotation $\mathcal{G}$ loops infinitely. The problem is that $\mathcal{G}$ encounters the progressively larger terms $rr\ l\ Nil$, $rr\ zs\ (Cons\ z_1\ Nil)$, $rr\ zs\ (Cons\ z_2\ (Cons\ z_1\ Nil))$, *etc.*

Since the formal parameter ys of rr is bound to progressively larger terms, Chin calls x an *accumulating parameter.*[7] We might also in the spirit of Chin call rr a *bad consumer* of its ys argument, because rr does not consume the value bound to ys as quickly as it is built up in the calls to rr.

Note that each of the problematic terms that are bound to ys is a subterm of the term which is subsequently bound to ys.

Let us agree that ys should be extracted, *i.e.* we put ys in rr's index set, and retain the remaining trivial annotations. Applying $\mathcal{G}$ to this term and program followed by postunfolding then simply yields the original program and term. This is satisfactory. □

EXAMPLE **3** (The Obstructing Function Call.)

$$\begin{array}{lcl} & & r\ l \\ r\ Nil & \leftarrow & Nil \\ r\ (Cons\ z\ zs) & \leftarrow & a\ (r\ zs)\ z \\ a\ Nil\ y & \leftarrow & Cons\ y\ Nil \\ a\ (Cons\ x\ xs)\ y & \leftarrow & Cons\ x\ (a\ xs\ y) \end{array}$$

The r function again reverses its argument, this time by first reversing the tail and then appending the head to this (the a function puts the element y in the end of its first argument.) Now the problem is that $\mathcal{G}$ encounters the terms $r\ l$, $a\ (r\ zs)\ z_1$, $a\ (a\ (r\ zs)\ z_2)\ z_1$, *etc.*

We call each of the calls to r in the redex position an *obstructing function call*, since they prevent the surrounding term from ever being transformed.[8] We might also in the spirit of Chin call r a *bad producer*, because it will never evaluate to a term with an outermost constructor that the surrounding a could consume.

[7] The same phrase is usually used for the programming style rr is written in.

[8] We differ slightly from the terminology of Chin here.

Note that each of the problematic terms that $\mathcal{G}$ encounters appears in the redex position of the subsequent problematic term.

Let us agree that r should be extracted, *i.e.* we associate r with $\ominus$ and retain the remaining trivial annotations. Applying $\mathcal{G}$ to this term and program followed by postunfolding then yields the original program and term. This is satisfactory. □

There is an obvious connection between the two examples: the programs compute the same thing but in different ways. Although the underlying problem in some sense is the same in both programs, it is expressed by different symptoms in the two examples.

4 Tree grammars

This section introduces some notation and terminology necessary to state our analysis. The reader may proceed to the next section and then refer back when necessary.

DEFINITION **5** (Grammar terms, *etc.* , tree grammars.) Let gt, ge, gr and go be generic variables ranging over *grammar terms, grammar (lazy evaluation) contexts, grammar redexes* and *grammar observables*, respectively.[9]

$$\begin{array}{lcl} gt & ::= & \bullet \mid c\ gt_1 \ldots gt_n \mid f\ gt_1 \ldots gt_n \mid g\ gt_0 \ldots gt_n \mid N \\ ge & ::= & [] \mid g\ ge\ gt_1 \ldots gt_n \\ gr & ::= & f\ gt_1 \ldots gt_n \mid g\ (c\ gt_{n+1} \ldots gt_{n+m})\ gt_1 \ldots gt_n \mid g\ \bullet\ gt_1 \ldots gt_n \\ go & ::= & c\ gt_1 \ldots gt_n \mid \bullet \end{array}$$

$\mathcal{N}$ and $\mathcal{GT}$ denote the set of all nonterminals and grammar terms, respectively. By a *tree grammar*[10] we mean a subset of $\mathcal{N} \times \mathcal{GT}$, which will be written $\{N_i \to gt_i\}_{i \in J}$. Each $N_i \to gt_i$ is called a *production*. □

As is evident, grammar terms *etc.* extend ordinary terms *etc.* by the presence of *nonterminals,* ranged over by N. Also, variables have been replaced by $\bullet$ which means "any variable." This in effect means that we are identifying terms which differ only in the names of variables.

[9] We shall use t, e, r, o instead of gt, ge, gr, go, and omit the qualifier "grammar" in front of "term," "context," *etc.* when no confusion is likely to arise.

[10] We often use the unqualified term *grammar* instead of tree grammar.

The unique decomposition property is preserved in a slightly modified form: given a grammar term t, exactly one of the following cases occur: 1) t is observable; 2) t can be decomposed uniquely into e, r such that $t \equiv e[r]$; 3) t can be decomposed uniquely into e, N such that $t \equiv e[N]$. So structural definitions use the extra case: $e[N]$.

Just as the notion of context is convenient for grabing hold of the redex and denoting the unfolding of the function call in the redex, we need some notation that allows us to pick an occurrence of a nonterminal and replace it with one of the right hand sides of the nonterminal in some grammar.

DEFINITION **6** (Replacement of nonterminals, $*$-derivation.) A *d-context*[11] is a grammar term, containing one occurrence of () at a place where another grammar term might have occurred. The expression $e(t)$ denotes the result of replacing the occurrence of () in e by t.

Given a grammar G, G^* is the smallest grammar satisfying the closure rules:

$$\frac{N \to t \in G}{N \to t \in G^*} \qquad \frac{N \to e(N') \in G^* \;\; N' \to t \in G}{N \to e(t) \in G^*}$$

The computation of an element in G^* will also be called a *derivation.* □

5 Idea and examples of the analysis

This section introduces informally the analysis as well as its use; the next section introduces both rigorously.

Given term t_0 and trivially annotated program p, the overall method runs as follows.

1. calculate a grammar G approximating the set of terms that $\mathcal{G}[\![\ t_0\ ]\!]$ encounters.
2. look in this grammar and calculate suitable annotations.
3. if no new $\ominus$'s and no new parameters for index sets are found, stop; otherwise replace the annotation by its lub with the new annotation and go to step 1.

Note that the annotations get strictly worse each time step 3 leads to step 1.

[11] What we have previously called *contexts* will henceforth be called *c-contexts.*

EXAMPLE 4 The first grammar that will be computed for the first of our programs is:

$$
\begin{array}{lcl}
N^0 & \to & r \bullet \mid N^r \mid \bullet \mid N^{ys} \\
N^r & \to & rr\ N^{xs}\ Nil \mid N^{rr,Nil} \mid N^{rr,Cons} \\
N^{rr,Nil} & \to & N^{ys} \\
N^{rr,Cons} & \to & rr \bullet (Cons \bullet N^{ys}) \mid N^{rr,Nil} \mid N^{rr,Cons} \\
N^{xs} & \to & \bullet \\
N^{ys} & \to & Nil \mid Cons \bullet N^{ys}
\end{array}
$$

In the grammar there is a nonterminal N^f for each f-function in the program, a nonterminal $N^{g,c}$ for each clause of the definition of every g-function in the program, a nonterminal N^v for every variable in the program and finally a start nonterminal N^0. (Nonterminals with no productions are not shown.)

The idea is that $N^v \to t \in G^*$ for every t that v is bound to during the course of $\mathcal{G}[\![\ t_0\ ]\!]$ and that $N^h \to t \in G^*$ if a call to h occurred in the redex and this call subsequently was transformed, in a number of steps, to t. Also, if a term bound to v ended up in the redex position and this term subsequently was transformed, in a number of steps, to t then $N^v \to t \in G^*$. If $\mathcal{G}[\![\ t_0\ ]\!]$ encounters t then $N^0 \to t \in G^*$.

The grammar G is computed in steps which simulate steps of the deforestation algorithm $\mathcal{G}$. In the example, $\mathcal{G}$ starts with the term rl, whereas the initial grammar is $\{N^0 \to r \bullet\}$. Then $\mathcal{G}$ moves to $rr\,l\,Nil$, whereas we compute from our grammar the new productions $N^0 \to N^r$, $N^r \to rr\ N^{xs}\ Nil$, $N^{xs} \to \bullet$. The first production recalls that we unfolded a call to r. The second just recalls the definition of r; this is the same production for all calls to r. The last production recalls the argument in the call to r. Calling the grammar computed so far G, it holds that $N^0 \to rr \bullet Nil \in G^*$.

Now $\mathcal{G}$ instantiates l to the patterns of rr and thereby encounters the terms $rr\,zs\,(Cons\,z\,ys)$ and Nil. We must add something to G representing this step. The basic idea is for the grammar construction in each step to compute G^*, where G is the grammar computed so far, and then unfold redexes on the right hand sides of productions for N^0. The problem with this is that $*$ loops infinitely. The solution is to compute less of $*$ and unfold calls on *all* right hand sides yielding at least all the right productions. For instance, in the current step it is enough to derive $N^r \to rr \bullet Nil$ from $Nr \to rr\ N^{xs}\ Nil$ and $N^{xs} \to \bullet$; unfolding in the grammar similarly to above then yields the new productions $N^r \to N^{rr,Nil}$, $N^r \to N^{rr,Cons}$,

$N^{rr,Nil} \to N^{ys}$, $N^{rr,Cons} \to rr \bullet (Cons \bullet N^{ys})$, $N^{ys} \to Nil$. It then holds that $N^0 \to Nil$ and $N^0 \to rr \bullet (Cons \bullet \bullet)$.

The question arises: just how much of $*$ should be computed in each step? We must avoid looping, but compute enough to get detailed knowledge of the flow. Given $N \to e[g\, N'\, t_1 \ldots t_n]$ we must replace N' by right hand sides of form N'', $\bullet$, $c\, t'_1 \ldots t'_k$ to see which of g's clauses will be used by $\mathcal{G}$. Given $N^0 \to N$ we must replace N by right hand side of form $N', c t_1 \ldots t_n$ to recall that $\mathcal{G}$ encunters the arguments of terms with outermost constructors.

As an example of a derivation which is not necessary, consider the situation after the last step above. The next terms encountered by $\mathcal{G}$ by the instantiation of l are $Cons\ z\ ys$ and $rr\ zs\ Cons\ z'\ (Cons\ z\ zs)$. The grammar contains among others the productions $N^0 \to N^r$, $N^r \to N^{rr,Cons}$, $N^{rr,Cons} \to rr \bullet (Cons \bullet N^{ys})$. Here we can simply unfold the right hand side of the last production to get $N^{rr,Cons} \to N^{rr,Nil}$, $N^{rr,Cons} \to N^{rr,Cons}$, $N^{ys} \to Cons \bullet N^{ys}$. It then holds that $N^0 \to Cons \bullet \bullet \in G^*$ and $N^0 \to rr \bullet Cons \bullet (Cons \bullet \bullet) \in G^*$. It was not necessary to derive anything from N^0, before unfolding, to get this.

To get the entire grammar we only need to explain the two last productions for N^0. Deriving $N^0 \to Cons \bullet N^{ys}$ before unfolding we get these. Why did we not simply compute from $N^{ys} \to Cons \bullet N^{ys}$ the two productions $N^{ys} \to \bullet$ and $N^{ys} \to N^{ys}$? Surely this would also yield $N^0 \to \bullet, N^0 \to N^{ys} \in G^*$. This is true, but it would also lead the analysis to believe that ys becoms bound to *e.g.* $\bullet$ (in fact it *does* in this case, but in general a variable bound to a term with an outermost constructor will not necessarily be bound to the components.) This will (again, in the general case) lead the analysis to some incorrect conclusions about which clauses of rr are used.

Now let us take a look at the final grammar. Recall that the problem in Example 2 was that rr was called with the progressively larger arguments Nil, $Cons\, z_1\, Nil$, $Cons\, z_2\, (Cons\, z_1\, Nil)$, *etc.* The formal parameter of rr is ys, and in fact $N^{ys} \to t \in G^*$ when t ranges over all these terms. Also recall that we noted in Example 2 that each problematic term was a subterm of the subsequent problematic term. Notice how well this is reflected by the production $N^{ys} \to Cons \bullet N^{ys}$.

The problem of the Accumulating Parameter is generally reflected by a production $N^v \to e(N^v) \in G^*$. In preventing $\mathcal{G}$ from looping the idea is to extract every variable v for which $N^v \to e(N^v) \in G^*$ where $e \neq ()$. Using this annotation and recalculating the grammars yields a grammar G with no $N^v \to e(N^v) \in G^*$ for $e \neq ()$. (This will not generally hold for the first

recalculated grammar.) □

EXAMPLE **5** (Grammar for the Obstructing Function Call.) The first grammar that will be computed for the second of our programs is:

$$\begin{array}{lcl}
N^0 & \to & r\ \bullet \mid N^{r,Nil} \mid N^{r,Cons} \mid Nil \mid N^y \\
N^{r,Nil} & \to & Nil \\
N^{r,Cons} & \to & a\ (r\ \bullet)\ \bullet \mid a\ N^{r,Nil}\ \bullet \mid a\ N^{r,Cons}\ \bullet \mid N^{a,Nil} \\
N^{a,Nil} & \to & Cons\ N^y\ Nil \\
N^{a,Cons} & \to & Cons\ \bullet\ (a\ \bullet\ N^y) \\
N^y & \to & Cons\ \bullet\ Nil
\end{array}$$

Recall that the problem in Example 3 was that the terms $r\ l$, $a\ (r\ zs)\ z_1$, $a\ (a\ (r\ zs)\ z_2\ z_1$, *etc.* were encountered, and in fact $N^{r,Cons} \to \mathcal{B}[\![\ t\ ]\!] \in G^*$ as t ranges over these terms. Also recall that we noted in Example 3 that each problematic term appeared in the redex position of the subsequent problematic term. Notice how well this is reflected by the production $N^{r,Cons} \to a\ N^{r,Cons}\ \bullet$.

The problem of the Obstructing Function call is reflected by a production $N^h \to e[N^h] \in G^*$ with $e \neq []$. In preventing $\mathcal{G}$ from looping, the idea is to extract every function h for which $N^h \to e[N^h] \in G^*$ with $e \neq []$. Using this annotation and recalculating the grammars yields a grammar G with no $N^h \to e[N^h] \in G^*$ with $e \neq []$. □

6 The analysis

This section states the actual analysis. An approximation similar to the present has appeared in [Jon87], which was indeed the main inspiration for this work. Other similar grammar constructions have appeared in [And86] for approximating Term Rewriting Systems and in [Mog88] for computing binding-time annotations for a self-applicable partial evaluator with partially static structures.

DEFINITION **7** Given grammar G, $G^\diamond$ is the smallest grammar satisfying the closure rules:

$$\frac{N \to t' \in G}{N \to t' \in G^\diamond} \qquad \frac{N \to e[N'] \in G^\diamond \quad N' \to t \in G}{N \to e[t] \in G^\diamond} \qquad \frac{N^0 \to N' \in G^\diamond \quad N' \to t \in G}{N^0 \to t \in G^\diamond}$$

where $e \neq []$ and $t \in \{N'', v, c\,t_1 \ldots t_n\}$. □

The $\diamond$-operator is an operator which is weak enough to be finitely computable but strong enough to get detailed knowledge during the computation of the grammar, see the preceding section.

DEFINITION **8** (The forgetful map.) Define $\mathcal{B}$ to be the function which maps a term t to the grammar term which arises by replacing all variables in t with $\bullet$. $\Box$

DEFINITION **9** (Grammar construction.)

$$
\begin{array}{lcl}
\mathcal{U}_{\mathcal{D}}[\![\, N \to \bullet \,]\!] & = & \{\} \\
\mathcal{U}_{\mathcal{D}}[\![\, N \to c\, t_1 \ldots t_n \,]\!] & = & \{N \to t_1, \ldots, N \to t_n\}, \text{ if } N \equiv N^0 \\
 & = & \{\}, \text{ otherwise} \\
\mathcal{U}_{\mathcal{D}}[\![\, N \to e[f\, t_1 \ldots t_n] \,]\!] & = & \{N \to e[N^f], N^f \to t^f[v_i^f := N^{v_i^f}]_{i=1}^n\} \\
 & & \cup \cup_{i=1}^n \{N^{v_i^f} \to t_i\} \\
\mathcal{U}_{\mathcal{D}}[\![\, N \to e[g\, (c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n] \,]\!] & = & \{N \to e[N^{g,c}], N^{g,c} \to t^{g,c}[v_i^{g,c} := N^{v_i^{g,c}}]_{i=1}^n\} \\
 & & \cup \cup_{i=1}^{n+m} \{N^{v_i^{g,c}} \to t_i\} \\
\mathcal{U}_{\mathcal{D}}[\![\, N \to e[g\, \bullet\, t_1 \ldots t_n] \,]\!] & = & \cup_{j=1}^k \{N \to e[N^{g,c_j}], \\
 & & N^{g,c_j} \to t^{g,c_j}[v_i^{g,c_j} := \bullet]_{i=n+1}^m [v_i^{g,c_j} := N^{v_i^{g,c_j}}]_{i=1}^n\} \\
 & & \cup \cup_{j=1}^k \cup_{i=1}^n \{N^{v_i^{g,c_j}} \to t_i\} \\
\mathcal{U}_{\mathcal{D}}[\![\, N \to e[N'] \,]\!] & \equiv & \{\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{U}_{\mathcal{G}}[\![\, N \to o \,]\!] & = & \mathcal{U}_{\mathcal{D}}[\![\, N \to o \,]\!] \\
\mathcal{U}_{\mathcal{G}}[\![\, N \to e[h^I_{\ominus}\, t_1 \ldots t_n] \,]\!] & = & \cup_{v_i^h \in I} \{N^0 \to t_i\} \cup \mathcal{U}_{\mathcal{D}}[\![\, N^0 \to h\, u_1 \ldots u_n[u_i := \bullet, u_j := t_j]_{v_i^h \in I, v_j^h \notin I} \,]\!] \\
 & & \cup \{N \to e[\bullet]\} \\
\mathcal{U}_{\mathcal{G}}[\![\, N \to e[h^I_{\oplus}\, t_1 \ldots t_n] \,]\!] & = & \cup_{v_i^h \in I} \{N^0 \to t_i\} \cup \mathcal{U}_{\mathcal{D}}[\![\, N \to e[h\, u_1 \ldots u_n[u_i := \bullet, u_j := t_j]_{v_i^h \in I, v_j^h \notin I}]
\end{array}
$$

$$
\begin{array}{lcl}
\overline{\mathcal{U}}_{\mathcal{G}}(G) & = & \cup_{N \to t \in G} \mathcal{U}_{\mathcal{G}}[\![\, N \to t \,]\!] \\
G^0_{\mathcal{G}} & = & \{N^0 \to \mathcal{B}[\![\, t_0 \,]\!]\} \\
G^{i+1}_{\mathcal{G}} & = & \overline{\mathcal{U}}_{\mathcal{G}}(G^{i\,\diamond}_{\mathcal{G}}) \cup G^i_{\mathcal{G}} \\
G^{\infty}_{\mathcal{G}} & = & \cup_{i=0}^{\infty} G^i_{\mathcal{G}}
\end{array}
$$

$\Box$

The same notational conventions are employed here as in the formulation of the deforestation algorithm in section 2. Note that $G^{\infty}_{\mathcal{G}}$ depends not only on the program p but also on the term t_0.

Now we can state the final algorithm.

DEFINITION **10** Given term t_0 and program p. Let a be the trivial annotation for p.

1. calculate $G = G^{\infty}_{\mathcal{G}}$.
2. Let a' be trivial except that it associates with $\ominus$ every h for which there exists N^h, $e \neq []$ such that $N^h \rightarrow e[N^h] \in G^*$, and it puts v in the proper index set of the function of which v is a formal parameter whenever there exists N^v, $e \neq ()$ such that $N^v \rightarrow e(N^v) \in G^*$.
3. if $a = a \sqcup a'$ stop; otherwise put $a = a \sqcup a'$ and go to step 1.

□

Notice how well the criteria for finding annotations correspond to the problems of the Accumulating Parameter and the Obstructing Function Call. Also note that the criteria for finding annotations are decidable.

7 Correctness and relation to Wadler and Chin's methods

In [Sor93a] we have described a slightly more complicated analysis for which the analogue of the following theorem is proved.

THEOREM **1** *For every term t_0 and trivially annotated program p the algorithm in definition 10 terminates with a set of annotations such that $\mathcal{G}[\![\, t_0 \,]\!]$ in the context of the annotated program terminates.* □

[Sor93a] also describes a number of improvements of the basic technique, one of which is called *tracing the top-level arguments.*[12] Recall that Wadler's original formulation of the deforestation algorithm was guaranteed to terminate only when applied to a *treeless program* and *treeless term.* In [Sor93a] we prove the analogue of the following theorem.

THEOREM **2** *(Treeless Theorem.) Let p be a treeless trivially annotated program and t an arbitrary term. Let G be the grammar computed from p and t using the extension of tracing top-level arguments. Then there is no $N^f \rightarrow e[N^f] \in G^*$ with $e \neq []$ and no $N^v \rightarrow e(N^v) \in G^*$ with $e \neq ()$.* □

[12] This improvement deals with certain nested function calls. These are not interesting for the application to Logic Programming.

This implies that no treeless program will receive annotations by our method. Hence, whenever Chin's method finds that no annotations are required so does our method. However, it may still be the case that when annotations *are* required, Chin's method finds better annotations than our method. This can be resolved by suitable (simple) modifications of our method. On the other hand there are programs which are non-treeless and hence requires annotations by Chin's method, which do require annotations by by our method, see [Sor93a].

8 Application to Logic Programming

First let us consider the problem of partial evaluation as described in [Gal93].

The termination problem in this framework (the *Basic Algorithm*) is divided into two problems: termination of (1) the unfolding rule and (2) the overall algorithm, where (1) is considered relatively easy—just like in our framework where $\mathcal{D}$ always makes exactly one unfolding step. As an instance of problem (2) [Gal93] considers the following example.

EXAMPLE **6** In the course of transforming the goal and predicate

$$\begin{array}{lll} & :- & r(L,R). \\ r(A,B) & :- & rr(A,B,nil). \\ rr(nil,K,K). & & \\ rr([X|Xs],Rs,Ys) & :- & rr(Xs,Rs,[X|Ys]). \end{array}$$

the partial evaluator encounters the infinite set of atoms $rr(Xs,B,[X_1])$, $rr(Xs,B,[X_2,X_1])$, *etc.* This looks very much like what we have called an accumulating parameter; in fact we have already seen that a related problem of an accumulating parameter occurs when one applies $\mathcal{G}$ to a functional version of r. □

The solution taken in [Gal93] to avoid infinite specialization like in the example above is to incorporate a function, *abstract*, which replaces atoms by new atoms of which the former are instances, *i.e.* it replaces terms by variables. This is essentially what $\mathcal{G}$ does when it extracts: it replaces a subterm by a variable. The main difference is that $\mathcal{G}$ blindly obeys annotations which are computed *before* the actual transformation, while the Basic Algorithm uses an on-line abstract interpretation as described in [Gal88].

Using a version of the Basic Algorithm with an *abstract* function which simply obeys annotations, one can devise a grammar construction approximating the flow of the Basic Algorithm, and use the resulting grammar to discover accumulating parameters as in r.[13] Below we outline such a construction.

Firstly, the grammar construction must simulate unfold steps, so let us assume that we are using an unfold rule which always unfolds one step, like in deforestation. Then by a straight-forward translation of our technique to logic programming languages we get the following grammar for the example above.

$$\begin{array}{lcl} N^0 & \to & r(\bullet,\bullet) \mid N^r \\ N^r & \to & rr(N^A, N^B, nil) \mid N^{rr,1} \mid N^{rr,2} \\ N^{rr,2} & \to & rr(\bullet, N^{Rs}, [\bullet|N^{Ys}]) \mid N^{rr,1} \mid N^{rr,2} \\ N^A & \to & \bullet \\ N^B & \to & \bullet \\ N^{Rs} & \to & N^B \mid N^{Xs} \\ N^{Ys} & \to & nil \mid [\bullet|N^{Ys}] \end{array}$$

This grammar reveals that Ys is an accumulating parameter. Specifically, we should define $\mathit{abstract}(r(\mathcal{X},\mathcal{Y},\mathcal{Z})) = r(\mathcal{X},\mathcal{Y},V)$, for arbitrary terms $\mathcal{X},\mathcal{Y},\mathcal{Z}$ in the logic programming language, and some variable V of the logic programming language.

While this all seems very well, there are in fact new problems to be solved when the analysis is to be conducted in a general logic programming language.

First, predicates are defined by multiple, nested patterns. It is easy to extend the grammar construction to account for that.

Secondly, predicates are defined by non-linear patterns. This introduces new problems, depending on the details of the specializer. As mentioned, non-linear patterns correspond essentially to equality tests (unifications, in the case of logic programming.) If the specializer propagates information about such tests, see [Sor93b], it may be desirable to represent this information in the grammar analysis as well. The succes of such a test is easy to represent by a substitution in the specializer and productions in the grammar construction, but a failure of a test is more complicated to represent (both in the specializer and in the grammar construction.)

[13] Incidentally, the problem of the obstructing function call does not seem to occur in [Gal93], because the unfolding rule collects all atoms in a *set*. This would correspond to all nested function calls being collected in a set in our framework.

Thirdly, recall that when the deforestation algorithm sees a term like $g\ v\ t_1 \ldots t_n$ where v occurs in a t_i, then the residual function also has v as a formal parameter. In other words, only the occurrence of v between g and t_1 is instantiated. However, in logic programming specializers, *all* the occurrences of a variable are instantiated. The grammar construction should be changed along the lines of the following description to account for this. We introduce a new kind of productions called, say, *i-productions.* Suppose that the grammar construction faces a goal which matches a predicate via an instantiatation, *e.g.* of X to $f(Y, Z)$ where X, Y, Z are variables. We should then introduce a production $N^X \rightarrow_i f(N^Y, N^z)$. As a criterion for infinity we should then also take $N^X \rightarrow_i e(N^X) \in G^*, e \neq ()$ where $N^X \rightarrow_i e(N^X)$ is derived using only i-productions and where N^{Xs} occurs on a right hand side of a non-i-production.

In closing this section, let us briefly consider the elimination procedure invented by Proietti and Pettorossi. Like the deforestation algorithm, this algorithm terminates for a certain class of programs only, *treelike* and *non-ascending* programs. This class is very similar to treeless functional programs.

Analogously to Chin's invention (but independently) Proietti and Pettorossi invented an algorithm using annotations, the *generalized elimination procedure*, and they devised a simple syntactic technique of calculating safe annotations which, again like Chin's technique, essentially annotates non-treelike and ascending atoms.

We expect that our technique can be used with benefit to calculate annotations for the elimination procedure as well.

9 Conclusion

We have presented a means of ensuring termination of deforestation which extends the amount of transformation that can be performed on programs, compared to Chin's previous method.

The method may also be suitable for ensuring termination of the elimination procedure and for partial evaluation in Logic Programming.

Acknowledgements. I would like to thank Chin for explaning details of his method. I would like to thank Robert Glück for comments on a previous version of this paper. Finally I would like to thank Neil Jones, my supervisor, for being a great inspiration. More formally I also owe Neil the credit that

the idea of using something similar to his grammar-based data-flow analysis in [Jon87] to ensure termination of deforestation was originally his; I merely pursued the idea.

References

[And86] Nils Andersen. *Approximating Term Rewriting systems With Tree Grammars.* DIKU-report 86/16, Institute of Datalogy, University of Copenhagen, 1986.

[Aug85] Lennart Augustsson. Compiling Lazy pattern-matching. In *Conference on Functional Programming and Computer Architecture.* LNCS 201, 1985.

[Bur77] R. M. Burstall & John Darlington. A Transformation system for Developing Recursive Programs. In *Journal of the ACM.* Vol. 24, No. 1. January 1977.

[Chi90] Wei-Ngan Chin. *Automatic Methods for Program Transformation.* Ph.D. thesis, Imperial College, University of London, July 1990.

[Chi93] Wei-Ngan Chin. Safe Fusion of Functional expressions II: Further Improvements. In *Journal of Functional programming.* 11, 1993. *To appear.*

[Fer88] A. B. Ferguson & Philip Wadler. When will Deforestation Stop?. In *1988 Glasgow Workshop on Functional Programming.* August 1988.

[Gal88] J.P. Gallagher, M. Codish & E.Y. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. In *New Generation Computing.* 6:159-186, 1988.

[Gal93] J.P. Gallagher. Tutorial on Specialisation of Logic Programs. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Copenhagen, Denmark, 1993.

[Hug90] John Hughes. Why Functional Programming Matters. In *Research topics in Functional Programming.* Ed. D. Turner, Addison-Wesley, 1990.

[Jon87] Neil D. Jones. Flow analysis of Lazy higher-order functional programs. In *Abstract Interpretation of Declarative Languages.* Eds. Samson Abramsky & Chris Hankin, Ellis Horwood, London, 1987.

[Mog88] Torben Mogensen. Partially Static Structures in a Self-applicable Partial Evaluator. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjrner & N. D. Jones, North-Holland, 1988.

[Pro90] M.Proietti & A. Pettorossi. Synthesis of Eureka Predicates for Develping Logic Programs. In *Proceedings of ESOP '90, LNCS 432.* pp305-325, Copenhagen, Denmark, 1990.

[Pro91] M.Proietti & A. Pettorossi. Unfolding - Definition - Folding, in this order for avoiding unnecessary variables in logic programs. In *Proceedings of PLILP '91, LNCS 528.* pp347-358, Passau, Germany, 1991.

[Sor93a] Morten Heine Sørensen. *A New Means of Ensuring Termination of Deforestation.* Student Project, DIKU, Department of Computer Science, University of Copenhagen, 1993.

[Sor93b] Morten Heine Sørensen, Robert Glück, Neil D. Jones. *Driving Generalizes Both Partial Evaluation and Deforestation.* Unpublished manuscript. (Working title, to appear), 1993.

[Tur90] David Turner. An overview of Miranda. In *Research topics in Functional Programming.* Ed. D. Turner, Addison-Wesley, 1990.

[Wad87] P. L. Wadler. Efficient compilation of pattern-matching. In *The implementation of Functional Programming Languages,*. Ed. S. L. Peyton Jones, Prentice-Hall, 1987.

[Wad88] P. L. Wadler. Deforestation: Transforming programs to eliminate trees. In *European symposium On programming (ESOP).* Nancy, France, 1988.