A Grammar-based Data-flow Analysis to Stop Deforestation

Morten Heine Sørensen

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

Abstract. Wadler's *deforestation* algorithm removes intermediate data structures from functional programs, but is only guaranteed to terminate for *treeless* programs. Chin has shown how one can apply deforestation to all first-order programs: *annotate* non-treeless subterms and apply the *extended* deforestation algorithm which essentially leaves annotated subterms untransformed. We develop a new technique of putting annotations on programs. The basic idea is to compute a finite grammar which approximates the set of terms that the deforestation algorithm encounters. The technique extends Wadler's and Chin's in a certain sense.

1 Introduction

Modern functional programming languages like 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.

Early work on automatic elimination of intermediate data structures from functional programs includes Turchin's *supercompiler* [Tur80, Tur82] and Wadler's *listless transformer* [Wad84, Wad85]. The latter eliminates intermediate *lists*, but later Wadler invented *deforestation* [Wad88, Fer88], which eliminates intermediate data structures in general, and proved that the deforestation algorithm terminates when applied to *treeless* programs.

A method for applying deforestation to all first-order programs was later described by Chin [Chi90, Chi94]. Inspired by Wadler's *blazed* deforestation algorithm [Wad88] he invented an *extended* deforestation algorithm which essentially leaves *annotated* subterms untransformed. The problem remains to calculate annotations ensuring that application of the extended deforestation algorithm to the annotated program terminates; as few subterms as possible should be annotated. Chin essentially annotates non-treeless subterms.

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.

The remainder is organized as follows. Section 2: the language we study. Section 3: the deforestation algorithm. Section 4: the two kinds of non-termination

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

that can arise during deforestation. Section 5: some notation for the analysis. Section 6: the idea of the analysis and how it works on the two examples from section 4. Section 7: the analysis technically. Section 8: an improvement of the basic method. Section 9: an explanation of the fact that our method extends the methods by Wadler and Chin; the improvement of Section 8 is necessary for this. Section 10: conclusion.

2 Language and notation

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

Definition1 Object language. Let c, v, f, g range over constructor names, variable names, f-function names, and g-function names, respectively. Let t, p, d range over terms, patterns, and definitions, respectively.

$$
\begin{array}{l}
t ::= v \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_n \mid g\ t_0 \ldots t_n \mid \text{ let } v = t \text{ in } t' \\
p ::= c\ v_1 \ldots v_n \\
d ::= f\ v_1 \ldots v_n \leftarrow t \\
\quad \mid \ g\ p_1\ v_1 \ldots v_n \leftarrow t_1 \\
\qquad\qquad \vdots \\
\qquad g\ p_k\ v_1 \ldots v_n \leftarrow t_k
\end{array}
$$

g-functions have one pattern matching argument, f-functions have none. We use h to range over functions which are either f- or g-functions.

We require that no variable occur more than once in a pattern and 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 *lazy evaluation*, like in Miranda.

As usual we state the deforestation algorithm by rule for rewriting terms. For this, 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 call-by-name 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.* If the term is a variable or has an outermost constructor, it is an *observable.*

Definition2 Context, redex and observable. Let e, r, o range over contexts, redexes, and observables, respectively.

$$
\begin{array}{l}
e ::= [] \mid g\ e\ t_1 \ldots t_n \\
r ::= f\ t_1 \ldots t_n \mid g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n \mid g\ v\ t_1 \ldots t_n \mid \text{ let } v = t \text{ in } t' \\
o ::= c\ t_1 \ldots t_n \mid v
\end{array}
$$

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 let-construct is not intended for the programmer; it is adopted in the language as an alternative to annotations. Instead of annotating the "dangerous" parts of a program and applying an extended deforestation algorithm which works conservatively on annotated subterms, we transform dangerous parts of the program into let-expressions and let the deforestation algorithm work conservatively on let-expressions.

3 The deforestation algorihtm

The following definition of $\mathcal{D}$ is essentially the deforestation algorithm in [Fer88]. After the algorithm we explain $\mathcal{D}$ along with the notation employed.

Definition 3 Deforestation algorithm, $\mathcal{D}$.

$$
\begin{array}{lll}
(1)\ \mathcal{D}[\![\ v\ ]\!] & \Rightarrow v & \\
(2)\ \mathcal{D}[\![\ c\,t_1 \ldots t_n\ ]\!] & \Rightarrow c\ (\mathcal{D}[\![\ t_1\ ]\!]) \ldots (\mathcal{D}[\![\ t_n\ ]\!]) & \\
(3)\ \mathcal{D}[\![\ e[f\,t_1 \ldots t_n]\ ]\!] & \Rightarrow f^{\square}\ u_1 \ldots u_l & \\
\quad \textbf{where} \quad f^{\square}\ u_1 \ldots u_l & \leftarrow \mathcal{D}[\![\ e[\ t^f\{v_i^f := t_i\}_{i=1}^n\ ]\ ]\!] & \\
(4)\ \mathcal{D}[\![\ e[g\ (c\,t_{n+1} \ldots t_{n+m})\,t_1 \ldots t_n]\ ]\!] & \Rightarrow f^{\square}\ u_1 \ldots u_l & \\
\quad \textbf{where} \quad f^{\square}\ u_1 \ldots u_l & \leftarrow \mathcal{D}[\![\ e[\ t^{g,c}\{v_i^{g,c} := t_i\}_{i=1}^{n+m}\ ]\ ]\!] & \\
(5)\ \mathcal{D}[\![\ e[g\ v\,t_1 \ldots t_n]\ ]\!] & \Rightarrow g^{\square}\ v\ u_1 \ldots u_l & \\
\quad \textbf{where} \quad g^{\square}\ p_1^g\ u_1 \ldots u_l & \leftarrow \mathcal{D}[\![\ e[t^{g,c_1}\{v_i^{g,c_1} := t_i\}_{i=1}^n]\ ]\!] & \\
 & \vdots & \\
\qquad\qquad g^{\square}\ p_k^g\ u_1 \ldots u_l & \leftarrow \mathcal{D}[\![\ e[t^{g,c_k}\{v_i^{g,c_k} := t_i\}_{i=1}^n]\ ]\!] & \\
(6)\ \mathcal{D}[\![\ \text{let } v = t \text{ in } t'\ ]\!] & \Rightarrow \text{ let } v = \mathcal{D}[\![\ t\ ]\!] \text{ in } \mathcal{D}[\![\ t'\ ]\!] &
\end{array}
$$

Notation.

The algorithm should be understood in the context of some program p. It is written in an informal meta-language; the symbol $\Rightarrow$ denotes evaluation in this language.

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 $v_{n+1}^{g,c}, \ldots v_{n+m}^{g,c}$ are the variables in the pattern (m depends on c).

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 $i = 1 \ldots n$.

The arguments $v, u_1 \ldots u_l$ in the calls in clauses (3)-(5) are the free variables occurring in e and the t_i's. In clause (5), v is also included among $u_1 \ldots u_l$ if it occurs in e or the t_i's. (In this case v occurs twice in the call[2]).

[2] This is actually a very important point; see [Sor94].

Chin's producer-consumer view of deforestation.

In clauses (3)-(5) of $\mathcal{D}$, the term t is transformed into a *residual call* to a new function.[3] This is a *fold* step. The right hand side of the new function is defined (before further transformation) to be t unfolded one step.[4] This is an *unfold* step and a *define* step. In clause (5) of $\mathcal{D}$, an *instantiation* step is performed as well.

The unfold steps propagate constructors towards the root of the term and thereby execute construction and subsequent destruction of intermediate structures. Consider, for instance, the term $g_2\,(g_1\,(f\,t_0)\,t_1)\,t_2$. Using the terminology of Chin [Chi90], the idea[5] is that the redex $f\,t_0$ through a number of unfoldings becomes a term with an outermost constructor, *produces* a constructor. This will allow the surrounding g-function g_1 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 g_2 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.

Folding to increase termination.

As is well-known, deforestation hardly ever terminates in the present formulation. The "problem" is that the same term may be encountered over and over again. Therefore we introduce a *folding scheme*. Each of the multiple recursive calls to $\mathcal{D}$ in clauses (2),(5),(6) determine a *branch* of transformation. We assume that when a term is encountered for the second time in the same branch then transformation terminates with the generation of the residual call. No new definition is introduced, since this was done the first time the term was encountered.

When a term is encountered which differs merely in the choice of variable names from a previously encountered term in the same branch, then we assume that transformation terminates with the generation of an appropriate residual call too. This can be explicated in notation in various ways, see [Fer88, Chi94].

Effeciency of residual programs.

There are some well-known problems in ensuring that the output program of $\mathcal{D}$ is at least as efficient, and preferably more efficient, than the input.

First, there is a problem concerning duplication of computation in the case of non-linear functions. This problem is well-known in deforestation as well as in partial evaluation. Therefore we shall not be concerned with this problem here.

Second, many calls and let-expressions in the residual program may be unfolded, improving efficiency. This phenomenon is also well known in deforestation and partial evaluation, so we do not go into details here.

---

[3] We assume that these new functions are collected somehow in a new program.

[4] Actually the left hand side as well as the unfolding of t should be renamed; we ignore the details.

[5] The following is the *desired* situation; the algorithm does not behave this well on all program, as we shall see.

4 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 $\mathcal{D}$ to each of the programs loops infinitely. We also show that with certain small changes in the programs, $\mathcal{D}$ does not loop infinitely. These changes are called *generalizations*.

The Accumulating Parameter.

Example 1. Consider the following program.

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

The r function returns its argument list reversed. Applied to this program and term $\mathcal{D}$ loops infinitely. The problem is that $\mathcal{D}$ 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*.[6] Note that each of the problematic terms that are bound to ys is a subterm of the term which is subsequently bound to ys.

It would seem that we can solve the problem if we could, somehow, make sure that $\mathcal{D}$ could not tell the difference between the different terms that are bound to ys. This can be achieved by transforming the program into:

$$\begin{array}{lll} & & r\,l \\ r\,xs & \leftarrow & \text{let } v = Nil \text{ in } rr\,xs\,v \\ rr\,Nil\,ys & \leftarrow & ys \\ rr\,(Cons\,z\,zs)\,ys & \leftarrow & \text{let } v = Cons\,z\,ys \text{ in } rr\,zs\,v \end{array}$$

Applying $\mathcal{D}$ to this term and program terminates with the same term and program as output. This is satisfactory.

The Obstructing Function Call.

Example 2. Consider the following term and program.

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

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 $\mathcal{D}$ 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.[7] Note that each of the problematic terms that $\mathcal{D}$ encounters appears in the redex position of the subsequent problematic term.

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

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

It would seem that we can solve the problem if we could, somehow, make sure that $\mathcal{D}$ could not tell the difference between the different terms that occur in the redex position. This can be achieved by transforming the program into: (the change in the *term* is actually not necessary)

$$\begin{array}{lll} & & \text{let } v = r\, l \text{ in } v \\ r\, Nil & \leftarrow & Nil \\ r\, (Cons\, z\, zs) & \leftarrow & \text{let } v = r\, zs \text{ in } a\, v\, z \\ a\, Nil\, y & \leftarrow & Cons\, y\, Nil \\ a\, (Cons\, x\, xs)\, y & \leftarrow & Cons\, x\, (a\, xs\, y) \end{array}$$

Applying $\mathcal{D}$ to this term and program terminates with the same term and program as output. This is satisfactory.

Generalizations.

Let $e()$ denote a term with exactly one occurrence of () at a place where a subterm could have occurred, and let $e(t)$ denote the result of substituting t for the occurrence of ().

Turning $e(h\, t_1 \ldots t_n)$ into $\text{let } v = t_i \text{ in } e(h\, t_1 \ldots t_{i-1}\, v\, t_{i+1} \ldots t_n)$ is called *generalization of h's i'th argument.* In Example 1, we generalized rr's second argument.

Turning $e(h\, t_1 \ldots t_n)$ into $\text{let } v = h\, t_1 \ldots t_n \text{ in } e(v)$ is called *generalization of the call to h.* In example 2, we generalized the call to r.

Generalizing should be thought of as annotating. Instead of putting funny symbols on our programs we instead use a distinct language construct.

5 Tree grammars

This section introduces some notation and terminology necessary to state our analysis. We first define tree grammars which are grammars where the right hand sides of productions are terms with nonterminals. The right hand sides will be called grammar terms, and there are also notions of grammar context, grammar redex, and grammar observable. In grammar terms we do not have variables; instead we have $\bullet$ informally signifying "any variable."

Definition 4 Grammar term, redex, observable. Tree grammar. Let t, r, o range over grammar terms, grammar redexes, and grammar observables, respectively.

$$\begin{array}{lll} t & ::= & \bullet \mid c\, t_1 \ldots t_n \mid f\, t_1 \ldots t_n \mid g\, t_0 \ldots t_n \mid \text{let } \bullet = t \text{ in } t' \mid N \\ e & ::= & [] \mid g\, e\, t_1 \ldots t_n \\ r & ::= & f\, t_1 \ldots t_n \mid g\, (c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n \mid g\, \bullet\, t_1 \ldots t_n \mid \text{let } \bullet = t \text{ in } t' \\ o & ::= & c\, t_1 \ldots t_n \mid \bullet \end{array}$$

$\mathcal{N}$ and $\mathcal{GT}$ denote the set of all nonterminals and grammar terms, respectively. By a *tree grammar* we mean a subset of $\mathcal{N} \times \mathcal{GT}$, which will be written $\{N_i \to t_i\}_{i \in J}$. Each $N_i \to t_i$ is called a *production.* A *finite tree grammar* is a finite subset of $\mathcal{N} \times \mathcal{GT}$. Our construction will always compute a finite tree grammar.

We shall omit the qualifier "grammar" in front of "term," "context," *etc.* when no confusion is likely to arise. Instead of tree grammars we also often use the unqualified term: grammars.

Definition 5. Define $\mathcal{F}$ to be the function which maps a term t to the grammar term which arises by replacing all variables by $\bullet$.

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. Let $e()$ denote a grammar term with exactly one occurrence of () at a place where another grammar term could have occurred, and let $e(t)$ denote the result of substituting t for the occurrence of ().

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

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

A production in G^* will also be called a *derivation.*

6 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 and program p, the overall method runs as follows.

1. Calculate a finite grammar G approximating the set of terms that $\mathcal{D}[\![\ t\ ]\!]$ encounters.
2. Look in this grammar and calculate suitable generalizations.
3. If no new generalizations were calculated stop; otherwise perform the generalizations on the term and program and go to step 1 with the new program and term.

Grammar for the Accumulating Parameter.

Example 3. Recall the program from Section 4 showing the problem of the Accumulating Parameter. The first grammar G that will be computed for this term and program is:

$$\begin{array}{ll} F^0 \to r\ \bullet \mid F^r \mid \bullet \mid V^{ys} & F^r \to rr\ V^{xs}\ Nil \mid F^{rr}_{Nil} \mid F^{rr}_{Cons} \\ V^{xs} \to \bullet & F^{rr}_{Nil} \to V^{ys} \\ V^{ys} \to Nil \mid Cons\ \bullet\ V^{ys} & F^{rr}_{Cons} \to rr\ \bullet\ (Cons\ \bullet\ V^{ys}) \mid F^{rr}_{Nil} \mid F^{rr}_{Cons} \end{array}$$

In the grammar there are three kinds of nonterminals which we explain in turn.

1. A start nonterminal F^0. If $\mathcal{D}[\![\ t\ ]\!]$ encounters t' then $F^0 \to \mathcal{F}[\![\ t'\ ]\!] \in G^*$.
2. A nonterminal F^f for each f-function in the program and a nonterminal F^g_c for each clause of the definition of every g-function in the program. If a call to h occurs in the redex of some term $e[h\ t_1 \ldots t_n]$ that $\mathcal{D}$ encounters, and this term in a number of steps is transformed to $e[t']$ by performing steps in the redex, then $F^h \to \mathcal{F}[\![\ t'\ ]\!] \in G^*$

3. A nonterminal V^v for every variable in the program. If t' is bound to v during the course of $\mathcal{D}[\![t]\!]$, then $V^v \to \mathcal{F}[\![t']\!] \in G^*$.

To understand that the grammar is a safe approximation of the set of terms that the positive supercompiler encounters, the first of these three results suffice. The two latter are never actually used in the reasoning, but they motivate several ideas, for instance the technique for computing generalizations from a grammar.

Recall that the problem in Example 1 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 $V^{ys} \to \mathcal{F}[\![t]\!] \in G^*$ when t ranges over all these terms. Also recall that we noted in Example 1 that each problematic term was a subterm of the subsequent problematic term. Notice how well this is reflected by the production $V^{ys} \to Cons\ \bullet\ V^{ys}$.

The problem of the Accumulating Parameter is generally reflected by a production $V^v \to e(V^v) \in G^*$. In preventing $\mathcal{D}$ from looping the idea is to generalize all arguments corresponding to some variable v for which $V^v \to e(V^v) \in G^*$ where $e \neq ()$. So we should generalize the second argument in all calls to rr. This yields the second program in Example 1. Recalculating the grammar for the new term and program yields a grammar G with no $V^v \to e(V^v) \in G^*$ for $e \neq ()$. (This will not generally hold for the first recalculated grammar).

The grammar is computed in steps which simulate steps of $\mathcal{D}$. The overall idea in computing the grammar G_∞, approximating the set of terms that $\mathcal{D}[\![t]\!]$ encounters, is as follows.

1. $G_0 = \{F^0 \to \mathcal{F}[\![t]\!]\}$
2. $G_{i+1} = \mathcal{E}(G_i^*) \cup G_i$, where $\mathcal{E}(H)$ is a certain function which looks at each right hand side of F^0 in H and computes certain new productions representing unfolding of terms.
3. $G_\infty = \cup_{i=0}^{\infty} G_i$

Recall point 1 in the meaning of productions: that $F^0 \to \mathcal{F}[\![t']\!] \in G_\infty^*$ for all t' that $\mathcal{D}$ encounters. The idea is that this follows from the invariant: $F^0 \to \mathcal{F}[\![t']\!] \in G_i^*$ for all t' encountered by $\mathcal{D}$ in at most i steps.

Below we show for the example program how each step of $\mathcal{D}$ is accompanied by an addition of productions to the grammar.

Step 1. $\mathcal{D}$ starts with the term $r\,l$, whereas the initial grammar G_0 is $\{F^0 \to r\ \bullet\}$. It then clearly holds that $F^0 \to \mathcal{F}[\![r\ l\ ]\!] \in G_0^*$.

Step 2. Then $\mathcal{D}$ moves to $rr\ l\ Nil$, whereas we add to our grammar the new productions

$$F^0 \to F^r, F^r \to rr\ V^{xs}\ Nil, V^{xs} \to \bullet$$

yielding the new grammar G_1. The first production records that we unfolded a call to r (in the empty context). The second just records the definition of r; this is the same production for all calls to r. The last production records the binding of a variable to xs. It now holds that $F^0 \to \mathcal{F}[\![rr\ l\ Nil]\!] \in G_1^*$.

This step shows the difference between nonterminals V^v and $\bullet$ in a right hand side. When we have $F^0 \to e(V^x)$ in the grammar then it means that $\mathcal{D}$ encounters the term $e(t)$ where t is some term that is bound to x at some point. When we have $F^0 \to e(\bullet)$ it means that $\mathcal{D}$ encounters the term $e(x)$ where x is some variable.

Step 3. Now $\mathcal{D}$ instantiates l to the patterns of rr and thereby encounters the terms Nil and $rr\ zs\ (Cons\ z\ Nil)$. We must add something to G_1 representing this step.

As mentioned above, the basic idea is for the grammar construction in the i'th step to compute G_i^*, where G_i is the grammar computed so far, and then unfold redexes on the right hand sides of productions for F^0. For instance, after step 1, we had $F^0 \to r \bullet \in G_0^*$, and in step 2 we computed G_0^*, and unfolded the right hand side $r \bullet$ to the new productions $F^0 \to F^r$, *etc.* The problem with this is that G_i^* may be infinite and so computation of it will loop infinitely. The solution is to compute from G_i only part of G_i^* and unfold calls on *all* right hand sides yielding at least all the right productions.

In the current step, for instance, we can derive $F^r \to rr \bullet Nil$ from the two last productions computed in step 2 and then unfold the right hand side $rr \bullet Nil$ of F^r to productions

$$F^r \to F^{rr}_{Nil}, F^r \to F^{rr}_{Cons}, F^{rr}_{Nil} \to V^{ys}, F^{rr}_{Cons} \to rr \bullet (Cons \bullet V^{ys}), V^{ys} \to Nil$$

yielding G_2. The first two productions record that we unfold the call to rr according to the different instantiations. The next two productions record the clauses of rr, and the last production records the binding caused by the unfolding. It now holds that $F^0 \to Nil, F^0 \to \mathcal{F}[\![\, rr\ zs\ (Cons\ z\ Nil)\ ]\!] \in G_2^*$.

The question arises: just how much of G_i^* should be computed in the i'th 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'', $\bullet$, and $c\ t'_1 \ldots t'_k$ to see which of g's clauses will be used by $\mathcal{D}$. Given $F^0 \to N$ we must replace N by right hand side of form N' or $c\ t_1 \ldots t_n$ to recall that $\mathcal{D}$ encounters the arguments of terms with outermost constructors.

Step 4. The next terms encountered by $\mathcal{D}$ by the instantiation of zs are $Cons\ z\ Nil$ and $rr\ zs\ Cons\ z'\ (Cons\ z\ Nil)$. The grammar G_2 contains among others the productions $F^0 \to F^r$, $F^r \to F^{rr}_{Cons}$, $F^{rr}_{Cons} \to rr \bullet (Cons \bullet V^{ys})$. Here we unfold the right hand side of the last production to get

$$F^{rr}_{Cons} \to F^{rr}_{Nil}, F^{rr}_{Cons} \to F^{rr}_{Cons}, V^{ys} \to Cons \bullet V^{ys}$$

yielding G_3. Here we did not add productions for the clauses of rr, since these were already present in G_2. It now holds that $F^0 \to \mathcal{F}[\![\ Cons\ z\ Nil\ ]\!] \in G_3^*$ and $F^0 \to \mathcal{F}[\![\ rr\ zs\ Cons\ z'\ (Cons\ z\ Nil)\ ]\!] \in G_3^*$.

Note that how little of G_2^* it was necessary to compute before unfolding right hand sides, to get this.

Step 5. In the next step $\mathcal{D}$ encounters (among others) the terms Nil and z. Deriving $F^0 \to Cons \bullet V^{ys}$ from G_3 and unfolding we get the two productions $F^0 \to \bullet$ and $F^0 \to V^{ys}$.

Why did we not simply compute from $V^{ys} \to Cons \bullet V^{ys}$ the two productions $V^{ys} \to \bullet$ and $V^{ys} \to V^{ys}$? Surely this would also yield $F^0 \to \bullet, F^0 \to V^{ys} \in G_4^*$. This is true, but it would also lead the analysis to believe that ys becoms bound to *e.g.* z. 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.

We have now arrived at the final grammar.

Grammar for the Obstructing Function Call

Example 4. Recall the program from Section 4 showing the problem of the obstructing function call. The first grammar that will be computed for this program is:

$$\begin{array}{ll}
F^0 \to r \bullet \mid F^r_{Nil} & F^r_{Nil} \to Nil \\
\quad \mid F^r_{Cons} \mid Nil \mid V^y & F^r_{Cons} \to a\ (r \bullet) \bullet \mid a\ F^r_{Nil} \bullet \mid a\ F^r_{Cons} \bullet \mid F^a_{Nil} \\
V^y \to \bullet & F^a_{Nil} \to Cons\ V^y\ Nil
\end{array}$$

Recall that the problem in Example 2 was that the progressively larger terms $r\,l$, $a\,(r\,zs)\,z_1$, $a\,(a\,(r\,zs)\,z_2\,z_1$, *etc.* were encountered, and in fact $F^0 \to \mathcal{F}[\![\, t \,]\!] \in G^*$ as t ranges over these terms. Also recall that we noted in Example 2 that each problematic term appeared in the redex position of the subsequent problematic term. Notice how well this is reflected by the production $F^r_{Cons} \to a\, F^r_{Cons}\, \bullet$.

The problem of the Obstructing Function call is reflected by a production $F^h \to e[F^h] \in G^*$ with $e \neq []$. In preventing $\mathcal{D}$ from looping, the idea is to generalize calls to every function h for which $F^h \to e[F^h] \in G^*$ with $e \neq []$. So we must generalize all calls to r. This yields the second program in Example 2. Recalculating the grammar for the new term and program yields a grammar G with no $F^h \to e[F^h] \in G^*$ with $e \neq []$.

7 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. Turchin has also used the idea of using grammars to approximate the terms that the supercompiler will encounter, see [Tur80] (Section 5.4). He uses the grammar approximation to get better transformation in the case of nested function calls, but apparently not to ensure termination.

Recall from Example 3 that the original idea was as follows: (*i*) compute a grammar, find generalizations, perform them on the term and program, and continue this cycle until no new generalizations are found. (*ii*) compute each grammar in a number of steps. (*iii*) in each of these steps first compute the grammar G^*_i (actually something less), and then (*iv*) apply a certain unfolding function on right hand sides of F^0 in G^*_i. Below we describe these steps precisely in the order: *(iii), (iv), (ii), (i)*. The four parts of the construction are stated in each their definition; all text not appearing within the definitions is merely explanatory.

Recall from Example 3 that we should not actually compute G^*_i and apply the unfolding function to all right hand sides of F^0, since G^*_i may be infinite. Instead we compute a smaller, finite, grammar $G^\diamond_i$ in each step and then apply the unfolding function to the right hand sides of *all* nonterminals, not just those of F^0.

We now define and explain $\diamond$. The reader familiar with [And86] may read it as an implementation of the notion of *minimal incorporation.*

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

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

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

The first rule says that all productions in G_i are in $G^\diamond_i$. If we have $F^0 \to t \in G_i$ where t is something giving rise to new productions, *e.g.* $e[r]$ where r is an f-function call, then, because $F^0 \to t \in G^\diamond_i$, we get these new productions by considering the productions in $G^\diamond_i$.

The second rule can be motivated as follows. Suppose that $F^0 \to g\ N \in G_i^\diamond$ and that $N \to c\ t_1 \ldots t_n$. Considering each of these two productions independetly does not tell the unfolding function anything. But given the information $F^0 \to g\ (c\ t_1 \ldots t_n) \in G_i^\diamond$ the unfolding function knows that a call to g will be encountered by $\mathcal{D}$ and that the call will be unfolded to the clause for c, and the unfolding function will generate new productions to account for this.

Why must the side condition on e, t hold in this rule? Well, *a priori* the idea is to get as *many* side conditions as possible, reducing the size of $G^\diamond$. The side condition is then motivated by the fact that these replacements of nonterminals by right hand sides suffice to maintain the invariant that $F^0 \to t \in G_i^*$ for all t encountered by $\mathcal{D}$ in at most i steps, and thereby $F^0 \to t \in G_\infty$ for all t encountered by $\mathcal{D}$ at all.

For instance, if G_i contains $F^0 \to g\ N, N \to f\ t$, then $*$ would compute $F^0 \to g\ (f\ t)$ before applying the unfolding function, the latter then yielding (roughly) the new productions (1) $F^0 \to g\ F^f, F^f \to t^f, V^{v^f} \to t$. But it is enough to apply the unfolding function right away to the production $N \to f\ t$ yielding (2) $N \to F^f, N^f \to t^f, N^{v^f} \to t$. The union of the productions (2) with G_i contains all the information that the union of (1) to G_i does.

One of the unfolding rules will state that if a term with an outermost constructer is encountered by $\mathcal{D}$, then the grammar must record that each of the constructor arguments also will be reached by $\mathcal{D}$. But it does not hold that if some function call appears in the redex and ends up being evaluated to $c\,t_1 \ldots t_n$ in the same context, then evaluation will proceed with the arguments in the redex; on the contrary, the redex position will change to the surrounding g-function call. Therefore, it is necessary to know for a right hand side with an outermost constructor whether it is derivable from F^0 or some other nonterminal; in the latter case $\mathcal{D}$ does not encounter the constructer term alone, it encounters it as a subterm of a bigger term. This is why the last rule is restricted to the nonterminal F^0.

Definition 8 Unfolding function.

$$
\begin{array}{ll}
\mathcal{E}[\![\ N \to \bullet\ ]\!] & = \{\} \\
\mathcal{E}[\![\ N \to c\ t_1 \ldots t_n\ ]\!] & = \{N \to t_1, \ldots, N \to t_n\},\ \text{if } N \equiv F^0; \\
 & = \{\},\ \text{otherwise} \\
\mathcal{E}[\![\ N \to e[f\ t_1 \ldots t_n]\ ]\!] & = \{N \to e[F^f], F^f \to t^f\{v_i^f := V_i^f\}_{i=1}^n\} \\
 & \quad \cup_{i=1}^n \{V_i^f \to t_i\} \\
\mathcal{E}[\![\ N \to e[g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n\ ]\!] & = \{N \to e[F_c^g], \\
 & \quad F_c^g \to t^{g,c}\{v_i^{g,c} := V_i^{g,c}\}_{i=1}^n\} \\
 & \quad \cup_{i=1}^{n+m} \{V_i^{v^{g,c}} \to t_i\} \\
\mathcal{E}[\![\ N \to e[g\ \bullet\ t_1 \ldots t_n]\ ]\!] & = \cup_{j=1}^k \{N \to e[F_{c_j}^g], \\
 & \quad F_{c_j}^g \to t^{g,c_j}\{v_i^{g,c_j} := \bullet\}_{i=n+1}^m \{v_i^{g,c_j} := N_i^{v^{g,c_j}}\}_{i=1}^n \\
 & \quad \cup_{i=1}^n \{V_i^{g,c_j} \to t_i\}\} \\
\mathcal{E}[\![\ N \to e[N']\ ]\!] & = \{\} \\
\mathcal{E}[\![\ N \to e[\ \text{let } \bullet = t \text{ in } t']\ ]\!] & = \{F^0 \to t, N \to e[t']\}
\end{array}
$$

The same notational conventions are employed here as in the formulation of the deforestation algorithm, see Section 3.

As was explained in Example 3, $\mathcal{E}$ simulates $\mathcal{D}$'s unfolding mechanism so as to maintain the invariant $F^0 \to \mathcal{F}[\![\ t\ ]\!] \in G_i^*$ for all t encountered by $\mathcal{D}$ in at most i steps. For instance from the three productions representing the unfolding

of a function call, one can recover the term that $\mathcal{D}$ comes to in one step from the original right hand side.

By an abuse of notation we also apply $\mathcal{E}$ to grammars as follows.

Definition 9 Steps of the grammar construction. The grammar for term t_0 and program p is:

$$\begin{aligned}\mathcal{E}(G) &= \cup_{N \to t \in G} \mathcal{E}[\![\, N \to t \,]\!] \\ G_0 &= \{F^0 \to \mathcal{F}[\![\, t_0 \,]\!]\} \\ G_{i+1} &= \mathcal{E}({G_i}^{\diamond}) \cup G_i \\ G_\infty &= \cup_{i=0}^{\infty} G_i\end{aligned}$$

Now we can state the final algorithm, which takes as input a program and a term and returns another program and term for which $\mathcal{D}$ terminates.

Definition 10. Given term t and program p.

1. Calculate $G = G_\infty$.
2. If there exists e, e', t' such that
   (a) $F^0 \to e'(F^h), F^h \to e[F^h], F^h \to t' \in G_\infty^*$ for some F^h and with $e \neq []$, then generalize all calls to h in t, p;
   (b) $F^0 \to e'(V^v), V^v \to e(V^v), V^v \to t' \in G_\infty^*$ for some V^v and with $e \neq ()$, then generalize all arguments for v in t, p.
3. If no generalizations were performed stop; otherwise go to step 1 with the new term and program.

In [Sor93] we have proved:

Theorem 11. *(i) the grammar computed in step 1 of the preceding algorithm is always finite (and can be computed in a finte number of steps). (ii) the criterion in step 2 of the preceding algorithm is decidable. (iii) for every term t and program p the preceding algorithm terminates with a term t' and program p' such that $\mathcal{D}[\![\, t' \,]\!]$ in the context of p' terminates.*

This settles the correctness issue. One might also wonder how *efficient* the analysis is. In a recent work [Sei93], Seidl reformulates the computation of the grammars of [Jon87, And86] as a normalization process of set equations, and shows that the procees can be carried out in polynomial time. We shall not be concerned with the complexity of our grammar analysis; future work may investigate whether it can be reformulated similarly to the constructions in [Jon87, And86].

8 A shortcoming and its solution

In this section we describe a shortcoming of the basic technique. A simple extension solves a certain, important class of instances of the problem, see the next section.

Example 5. Consider the following uninteresting term and program.

$$\begin{array}{ll} & f\ (f'\ z) \\ f'\ w & \leftarrow f\ w \\ f\ v & \leftarrow g\ v \\ g\ (C\ u) & \rightarrow C\ u \end{array}$$

The first two steps in transformation are: $\mathcal{D}[\![\ f\ (f'\ z)\ ]\!] \Rightarrow \mathcal{D}[\![\ g\ (f'\ z)\ ]\!] \Rightarrow \mathcal{D}[\![\ g\,(f\,z)\ ]\!]$. After a few more steps the transformation terminates. The grammar computed from this term and program is:

$$\begin{array}{lll} F^0 \to f\ (f'\ \bullet) \mid F^f \mid V^u & & F^{f'} \to f\ V^w \mid F^f \\ V^w \to \bullet & & F^f \to g\ V^v \mid F^g_C \\ V^v \to f'\ \bullet \mid F^{f'} \mid V^w & & F^g_C \to C\ V^u \mid C\ \bullet \\ V^u \to V^u & & \end{array}$$

Although there is no termination problem, we have $F^f \to g\ F^f \in G^*$ signifying that calls to f should be generalized. We have cheated the grammar construction by having an initial term $f\ t$ which unfolds to $e[f\ t']$ where the second call to f is not "caused" in any way by the first.

The solution to this shortcoming is to make sure that $F^f \to g\ F^f \in G^*$ only happens when the second call really is caused by the first in some way. This can be done by a theoretically simple extension as follows. Suppose that we wish to transform the term t in program p and that t contains, syntactically, n function calls. Take n copies of p. In the i'th copy, all function names and variable names are subscripted with i. In the term, subscript each function call with a unique number i between 1 and n, informally signifying that this call be evaluated in the i'th copy of p. Then compute the grammar. Now calculate generalizations from this grammar as usual. For instance, $F^{f_1} \to e[F^{f_2}] \in G^*$ does not signify infinity but $F^{f_1} \to e[F^{f_1}] \in G^*$ does. Perform generalizations on the n copies of the program accordingly, and transform the term using the n copies of the program.

Note that there is nothing new in this method, theoretically. Before transformation, we simply make n copies of the program, and change the original calls in the term into calls to the different versions. One can avoid actually making n copies of the program, see [Sor93].

Example 6. For the example, the first steps of the transformation are: $\mathcal{D}[\![\ f_1(f'_2 z)\ ]\!] \Rightarrow \mathcal{D}[\![\ g_1\ (f'_2\ z)\ ]\!] \Rightarrow \mathcal{D}[\![\ g_1\ (f_2\ z)\ ]\!]$. Here we see the important point: now the two calls to f are separable—they have different numbers. Indeed, the computed grammar for this new term and program does not imply any generalizations.

9 Relation to Wadler and Chin's methods

Recall that Wadler's original formulation of the deforestation algorithm was guaranteed to terminate only when applied to a *treeless program*. In [Sor93] we prove the following proposition.

Proposition 12. *Let p be a treeless program and t an arbitrary term. Let G be the grammar computed from p and t using the extension from the preceding section. Then there is no $F^h \to e[F^h] \in G^*$ with $e \neq []$ and no $V^v \to e(V^v) \in G^*$ with $e \neq ()$.*

This implies that no treeless program will receive generalizations by our method. Hence, whenever Chin's method finds that no annotations are required, our method finds no generalizations.

On the other hand there are programs which are non-treeless and hence requires annotations by Chin's method, which do require generalizations by our method.

Example 7. Consider the (rather contrived) term and program:

$$
\begin{array}{ll}
 & f\ v \\
f\ x & \leftarrow f'\ (C\ x) \\
f'\ y & \leftarrow g\ y \\
g\ (C\ z) & \leftarrow f\ z
\end{array}
$$

The reader may care to apply $\mathcal{D}$ to the term and verify that the process does not loop infinitely. The grammar is:

$$
\begin{array}{ll}
F^0 \rightarrow f\ \bullet \mid F^f & F^f \rightarrow f'\ (C\ V^x) \mid F^{f'} \\
V^y \rightarrow c\ V^x & F^{f'} \rightarrow g\ V^y \mid F^g_C \\
V^x \rightarrow \bullet \mid V^z & F^g_C \rightarrow f\ V^z \mid F^f \\
V^z \rightarrow V^x &
\end{array}
$$

The grammar does not imply any generalizations. We can see that the analysis reasons as follows. First f calls f' (*c.f.* $F^f \rightarrow F^{f'}$). In the program we can see that the argument is larger than that which f itself received. Second, f' calls g (*c.f.* $F^{f'} \rightarrow F^g$). The argument is the same as that which f' received. Finally, g calls f (*c.f.* $F^g \rightarrow F^f$). Here the argument is smaller than that which g itself received. The point is: will the argument to f get bigger and bigger in each incarnation of f, or is the decrement by g significant enough to outweigh the increment by f? Well, in the grammar we see that our analysis discovered that the argument that g binds to z when it is called from f' is the argument which f originally received (*c.f.* $V^z \rightarrow V^x$) and g then calls f with that same argument (*c.f.* $V^x \rightarrow V^z$), and this does not yield progressively larger arguments.

Note that the program is non-treeless. Chin's method would classify f' as an unsafe consumer, and annotate the argument Cx. This would in fact prevent the C in the call to f' from being eliminated by g. This shows that the grammar can detect *decrements that outweigh increments*. Our construction can also handle other violations against the treeless form.

10 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.

Acknowledgements.

I would like to thank the Wei-Ngan Chin for explaining me details of his method and for many comments to [Sor93] out of which this paper grew. Also thanks to Nils Andersen, David Sands, and Robert Glück for comments and discussions. I would like to thank Neil Jones, my supervisor, for being a great inspiration and for suggesting many improvements in [Sor93]. 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. Last of all, I should express my appreciation for the patience of my (non-computer scientist) girl-friend Mette Bjørnlund, who has come to know a lot about deforestation during my writing of this paper.

References

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

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

[Chi94] Wei-Ngan Chin. Safe Fusion of Functional expressions II: Further Improvements. In *Journal of Functional programming. To appear.*

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

[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.

[Sei93] Helmut Seidl. *Approximating Functional Programs in Polynomial Time.* Unpublished manuscript. 1993.

[Sor93] Morten Heine Sørensen. *A New Means of Ensuring Termination of Deforestation.* Student Project 93-8-3, DIKU, Department of Computer Science, University of Copenhagen, 1993.

[Sor94] Morten Heine Sørensen, Robert Glück, Neil D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In *European Symposium On Programming'94.* To appear as LNCS, 1994.

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

[Tur80] Valentin F. Turchin. *The Language REFAL—The Theory of Compilation and Metasystem Analysis.* Courant Computer Science Report 20, 1980.

[Tur82] Valentin F. Turchin, Robert M. Nirenberg, Dimitri V. Turchin. Experiments with a Supercompiler. In *ACM Symposium on Lisp and Functional Programming.* New york, 1982.

[Tur88] Valentin F. Turchin. The Algorithm of Generalization in the Supercompiler. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjrner & N. D. Jones North-Holland, 1988.

[Wad84] Philip Wadler. Listlessness is better than lazyness: Lazy evaluation and garbage collection at compile-time. In *ACM Symposium on Lisp and Functional Programming.* Austin, Texas, 1984.

[Wad85] Philip Wadler. Listlessness is better than lazyness II: Composing Listless functions. In *Workshop on Programs as Data objects.* LNCS 217, Copenhagen, 1985.

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

This article was processed using the LaTeX macro package with LLNCS style