Integer Constraints to Stop Deforestation

Helmut Seidl

FB IV – Informatik
University Trier
D–54286 Trier
seidl@ti.uni-trier.de

Abstract. Deforestation is a transformation of functional programs to remove intermediate data structures. It is based on outermost unfolding of function calls where folding occurs when unfolding takes place within the same nested function call. Since unrestricted unfolding may encounter arbitrarily many terms, a termination analysis has to determine those subterms where unfolding is possibly dangerous. We show that such an analysis can be obtained from a control flow analysis by an extension with integer constraints – essentially at no loss in efficiency.

1 Introduction

The key idea of flow analysis for functional languages is to define an abstract meaning in terms of *program points*, i.e., subexpressions of the program possibly evaluated during program execution [Pa95]. Such analysises have been invented for tasks like type recovery [Sh91], binding time analysis [Co93], or safety analysis [PS95]. Conceptually, these are closely related to A. Deutsch's store–based alias analysis [De90] or N. Heintze's method of approximating the collecting semantics by means of set constraints [He94]. N. Heintze especially observed that a formulation by constraints allows for a derivation of very fast approximation algorithms. It is our aim to enlarge the area of applications of this type of analysis by extending it with additional constraint systems. We examplify our idea with a problem related to deforestation.

In 1988, Phil Wadler came up with an algorithm to eliminate intermediate data structures from first order functional programs [Wa88]. Recently, this optimizing transformation gained additional attraction in the context of program specialization and partial evaluation [SGJ94, NS95]. As an example, consider the following program:

```
letrec
    flip (Leaf x)  = Leaf x
    flip (Node y1 y2)  = Node (flip y2) (flip y1)
in
    flip (flip t)
```

Deforestation starts with the term `flip (flip t)`. It performs a sequence of outermost unfoldings where for every expression of nested function calls to be unfolded, a new function definition is generated:

$$
\begin{aligned}
D\llbracket\texttt{flip (flip t)}\rrbracket &\Rightarrow f_1\ \texttt{t}\\
&\textbf{where}\\
f_1(\texttt{Leaf x}) &= D\llbracket\texttt{flip (Leaf x)}\rrbracket\\
&\Rightarrow f_2\ \texttt{x}\\
f_1(\texttt{Node y1 y2}) &= D\llbracket\texttt{flip (Node (flip y2) (flip y1)}\rrbracket\\
&\Rightarrow f_3\ \texttt{y1 y2}\\
&\textbf{where}\\
f_2\ \texttt{x} &= D\llbracket\texttt{Leaf x}\rrbracket\\
&\Rightarrow \texttt{Leaf x}\\
f_2\ \texttt{y1 y2} &= D\llbracket\texttt{Node (flip (flip y1)) (flip (flip y2))}\rrbracket\\
&\Rightarrow \texttt{Node}\ D\llbracket\texttt{flip (flip y1)}\rrbracket\ D\llbracket\texttt{flip(flip y2)}\rrbracket
\end{aligned}
$$

Observe that whenever the function of the function call to be unfolded is defined by case distinction on the topmost constructor of some argument and the corresponding actual parameter is just a free variable, then the newly introduced function is also defined using pattern matching on this argument.

Unfolding stops whenever (an instance of) the same expression is going to be unfolded twice (in our example `flip(flip t)`). In this case, it is folded to a call of the corresponding new function (in the example to (f_1 `y1`) and (f_1 `y2`), respectively).

In [Sa95], D. Sands proves that (even the higher order extension of) this transformation is indeed correct. Unrestricted unfolding, however, may lead to an arbitrarily growing number of new functions. Two typical examples are given by the two implementations of the reverse function for lists given in Fig. 1 and 2 (cf. [Sø94b]). In the version of Fig. 1, deforestation is applied to the increasing sequence of terms `rr zs` (z_1`:[]`), `rr zs` (z_2`:`z_1`:[]`), `rr zs` (z_3`:`z_2`:`z_1`:[]`), ... This is called the *accumulating parameter* problem by Chin and Sørensen. On the other hand consider the implementation of reverse according to Fig. 2. Now the sequence `a (r` xs`)` x_1, `a (a (r` xs`)` x_2`)` x_1, `a (a (a (r` xs`)` x_3`)`

```
letrec
    r xs        = rr xs []
    rr [] y     = y
    rr (z:zs) y = rr zs (z:y)
in
    r l
```

Fig. 1. The reverse function with accumulating parameter

```
letrec
    r []          = []
    r (x:xs)      = a (r xs) x
    a [] y        = y:[]
    a (z:zs) y    = z:(a zs y)
in
    r l
```

Fig. 2. The reverse function with obstructing function call

$x_2)\ x_1, \ldots$ of arbitrarily nested function calls is encountered. This was called the *obstructing function call* problem.

One idea to deal with such unappreciated behavior is to mark certain subterms as potentially dangerous and ignore these during the deforestation. This idea, proposed already in [Wa88] has been elaborated by Chin in [Ch90, Ch94]. In [Sø94b] Sørensen presented a nice analysis to detect potentially dangerous terms. The analysis consists of two steps. First, the unfolding process of deforestation is approximated by a tree grammar. Secondly, necessary conditions are formulated for accumulating parameters resp. obstructing function calls to occur. By showing that (extended) treeless programs pass his test [1] he could show that his method results in an improvement of the original method.

We recast Sørensen's analysis by means of constraint systems of very simple forms. The basic one, operating with a finite set of terms, is used for abstracting the control flow of unfolding. The iteration process of computing the set of constraints for control flow analysis is used to produce a system of integer contraints. By this approach we achieve:

- a much simpler presentation;
- a more informative analysis: besides classifying function calls or argument positions as potentially dangerous or harmless, it also gives information about possible *depthes* of expressions encountered during transformation[2];
- an easy correctness proof;
- a simple and efficient implementation with polynomial runtime.

We also conjecture that generalizations to an analysis of more general transformations become easier.

In this paper we concentrate on the analysis itself. For detailed information on deforestation and its relation to program optimization and partial evaluation, consult, e.g., [Wa88, Sø94a, GJS94]. The paper is organized as follows. First, we shortly formalize the deforestation transformation. Next we introduce the constraint systems we are going to use. Then we present the analysis. A proof

[1] at least provided the functions occurring in the main expression are pairwise distinct; this, however, may w.l.o.g. always be assumed.

[2] A slight modification of the analysis presented here, could also compute upper bounds to the *sizes* of occurring expressions.

of it's correctness follows. Finally, we discuss the kind of information obtained by the analysis.

2 Deforestation

For simplicity, we consider just a first order language whose expressions consist of terms t built up from variables (ranged over by x) and constants by constructor and function applications. We distinguish two kinds of functions: those (ranged over by f) which are defined without pattern matching and those (ranged over by g) with pattern matching on the top constructor of their 0–th arguments. h is meant to range over both constructor and function names.

$$\begin{array}{rlll} t ::= & & x & \text{(variable)} \\ & \mid & c\ t_1 \ldots t_n & \text{(constructor application)} \\ & \mid & f\ t_1 \ldots t_n & \text{(f–function application)} \\ & \mid & g\ t_0 \ldots t_n & \text{(g–function application)} \\ & \mid & \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 & \text{($\mathbf{let}$ construct)} \end{array}$$

As in [Sø94b], the **let**–construct is used to mark dangerous subterms.

Assume t is not a constructor application. Then t uniquely can be decomposed into $t = e[r]$ where r is the subterm of t which by CBN reduction strategy is going to be reduced next and e is its *reduction context*. Formally, reduction contexts are defined by:

$$e ::= [] \mid g\ e\ t_1 \ldots t_n$$

The depth $depth(t)$ of a term t is defined as usual where nullary symbols have depth 0. For reduction contexts we define their depth by $\text{depth}([]) = 0$ and $\text{depth}(g\ e\ t_1 \ldots t_n) = 1 + \text{depth}(e)$.

A program p in this language consists of a sequence of (possibly) mutually recursive function definitions together with one main expression *main*.

$$\begin{array}{rl} p ::= & \mathbf{letrec}\ \mathit{defs}\ \mathbf{in}\ \mathit{main} \\ \mathit{main} ::= & t \\ \mathit{defs} ::= & \epsilon \mid \\ & \quad f\ x_1 \ldots x_n = t\ ;\ \mathit{defs} \mid \\ & \quad g\ (c\ x_{n+1} \ldots x_m)\ x_1 \ldots x_n = t\ ;\ \mathit{defs} \end{array}$$

For convenience, we denote the right hand side of function f by t^f and its formals by x_i^f. Accordingly, the right hand side of function g corresponding to constructor c is denoted by $t^{g,c}$ whereas its formal parameters are $x_1^g, \ldots, x_n^g$ together with variables $x_{n+1}^{g,c}, \ldots, x_m^{g,c}$ from the pattern.

The deforestation procedure $\mathcal{D}$ is based on outermost unfolding of terms possibly containing free variables. Besides the term it is applied to, a call of $\mathcal{D}$ receives as an additional argument the set of currently existing function definitions. Accordingly, its result consists of a term together with a possibly enlarged set of function definitions. It is for simplicity of the presentation that we omit making the bookkeeping on function definitions explicit. Thus, we leave the set of already existing function definitions before the call implicit and only indicate possible newly created function definitions by means of a **where**–construct.

$$
\begin{array}{ll}
(0) & \mathcal{D}\llbracket x \rrbracket \Rightarrow x; \\
(1) & \mathcal{D}\llbracket c\ t_1 \ldots t_n \rrbracket \Rightarrow c\ \mathcal{D}\llbracket t_1 \rrbracket \ldots \mathcal{D}\llbracket t_n \rrbracket; \\
(2) & \mathcal{D}\llbracket e[f\ t_1 \ldots t_n]\ \rrbracket \Rightarrow f^{\square} u_1 \ldots u_l \\
 & \qquad\qquad\qquad\qquad \mathbf{where} \\
 & \qquad\qquad\qquad\qquad f^{\square} u_1 \ldots u_l = \mathcal{D}\llbracket e[t^f[t_i/x_i^f]\ ]\ \rrbracket; \\
(3) & \mathcal{D}\llbracket e[g\ (c\ t_{n+1} \ldots t_m) t_1 \ldots t_n]\ \rrbracket \Rightarrow f^{\square} u_1 \ldots u_l \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{where} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad f^{\square} u_1 \ldots u_l = \mathcal{D}\llbracket e[t^{g,c}[t_i/x_i^{g,c}]\ [t_j/x_i^g]\ ]\ \rrbracket; \\
(4) & \mathcal{D}\llbracket e[g\ (x\ t_1 \ldots t_n]\ \rrbracket \Rightarrow g^{\square} x u_1 \ldots u_l \\
 & \qquad\qquad\qquad\qquad\quad \mathbf{where} \\
 & \qquad\qquad\qquad\qquad\quad \ldots \\
 & \qquad\qquad\qquad\qquad\quad g^{\square}(c\ u_{l+1} \ldots u_{m'}) u_1 \ldots u_l = \mathcal{D}\llbracket e[t^{g,c}[u_{l+j}/x_{n+j}]\ [t_i/x_i^g]\ ]\ \rrbracket \\
 & \qquad\qquad\qquad\qquad\quad \ldots; \\
(5) & \mathcal{D}\llbracket e[\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2]\ \rrbracket \Rightarrow \mathbf{let}\ x = \mathcal{D}\llbracket t_1 \rrbracket\ \mathbf{in}\ \mathcal{D}\llbracket e[t_2]\ \rrbracket;
\end{array}
$$

Case (1) indicates that deforestation is pushed down through outermost constructors; case (2) describes unfolding of f–functions whereas cases (3) and (4) describe unfolding of g–functions. In case (3) the outermost constructor of the 0–th argument is known; therefore the resulting new function is an f–function. In case (4) however, the 0–th argument is a variable and therefore not known at transformation time. Hence, the resulting new function is a g–function. In all cases (2), (3) and (4), the $u_1, \ldots, u_l$ are (an enumeration of) the free variables of the unfolded expression. Finally, case (5) shows how the **let** construct is treated: deforestation proceeds to the subterms while ignoring possible interactions between t_1 and t_2.

3 Constraint Systems

Systems of set constraints are a convenient tool for describing (tuples of) regular tree languages. Various formalisms of different expressiveness have been considered in the literature. For the analysis of functional languages, N. Heintze considered only constraints of the form $X \supseteq e$ where the left hand side S simply consists of a variable and the right hand side is an expression built up from variables and nullary constructors by means of constructor applications only [Ne94]. This formalism is closely related to regular tree grammars where constraint $X \supseteq e$ is viewed as a rewrite rule $X \to e$. Since long, tree grammars have been used by N.D. Jones [Jo79, Jo87]. The advantage of the constraint formalism, however, lies in its simplicity. Also it gave rise to not only theoretically fast but also practical analysis algorithms. Here is an example of such a system.

$$Y \supseteq X \qquad Y \supseteq aXY \qquad X \supseteq b$$

A model ρ of such a system S assigns sets of terms to every variable such that all constraints in S are satisfied, i.e., $(\rho\ X) \supseteq [\![e]\!]\ \rho$ for every constraint $X \supseteq e$ in S. It is well–known that, provided all language operators occurring in right hand sides are monotonous, a least model exists. In our example, the least model μS assigns a regular tree language to every variable, namely $\mu S\ X = \{b\}$ and $\mu S\ Y = \{abb, ab(abb), ab(ab(abb)), \ldots\}$.

Constraint systems allowing for least models can be defined in greater generality. Let D denote a complete lattice. For some variable set *Vars*, we consider sets S of constraints of the form

$$X \sqsupseteq f X_1 \ldots X_n$$

where $X, X_1, \ldots, X_n \in \mathit{Vars}$, and f denotes a monotonous function $[\![f]\!] : D^n \to D$. Then for every such system S there is a least model μS mapping variables to elements of D such that $\mu S\ X \sqsupseteq [\![f]\!](\mu S\ X_1) \ldots (\mu S\ X_n)$ for every constraint $X \sqsupseteq f X_1 \ldots X_n \in S$. Furthermore note, that if S_1, S_2 are sets of constraints over the same set of variables *Vars* then $S_1 \subseteq S_2$ implies $\mu S_1\ X \sqsubseteq \mu S_2\ X$ for every variable $X \in \mathit{Vars}$. The constraint systems we are interested in here are:
1. Simple Constraints: These are even more trivial than set constraints. A finite set A of basic objects is given. D is defined as the powerset of A ordered by set inclusion; right hand sides may contain constants $a \in A$, variables or application of the intersection operation "$\cap$".

Example 1. For $A = \{a, b\}$, consider the system:

$$\begin{array}{ll} Y \supseteq b & \quad X \supseteq X \cap Y \\ Y \supseteq X & \quad X \supseteq a \end{array}$$

The least model of this example maps X to $\{a\}$ and Y to $\{a, b\}$.

In our analysis we only need constraints without occurrences of "$\cap$". Constraint systems of this type have been considered for Control Flow Analyis, e.g., in [Pa95, PO95].

In our analysis, we additionally need the special case where A contains just one element, say 1. In this case, 2^A is isomorphic to the 2–point domain $\mathbf{2} = \{\bot \sqsubset \top\}$. It turns out that adding such constraints is a convenient tool to abstract special reduction strategies. For modelling CBN, it is sufficient to consider constraints of the form $X \sqsupseteq 1$. In case, more complicated reduction strategies are studied, also more complicated constraints might be reasonable.
2. Integer Constraints: Here, complete lattice $D = \mathcal{N}$ consists of the nonnegative integers equipped with their natural ordering and extended by ∞. Right hand sides are built up from variables and constants by means of operators "$+$" and "$\sqcap$" (minimum).

Example 2.

$$\begin{array}{ll} X \geq 1 & \quad Y \geq X + Y \\ Z \ \geq X & \quad Z \geq Y \sqcap X \end{array}$$

The least model of this system is given by $X \mapsto 1, Y \mapsto \infty, Z \mapsto 1$.

Note that $\mathcal{N}$ does not satisfy the ascending chain condition. Therefore, an approach using naive fixpoint iteration may not suffice to compute the least model. In [Se94] it is shown that this least model nevertheless can be computed[3]. Various algorithms are considered which compute the least model efficiently for different forms of systems. In general, the least model can be computed in time $O(|S|^2)$ where $|S|$ denotes the size of constraint system S. In case, no minimum operator occurs even time $O(|S|)$ suffices.

4 The Analysis

The key observation for the construction of the analysis is that we can ignore folds. We consider arbitrary sequences of outermost unfoldings of terms possibly containing free variables. This unfolding is approximated by a control flow analysis. What we are aiming to guarantee is that during unfolding only finitely many expressions $e[h\ t_1 \dots t_n]$ are encountered. Instead of counting different expressions we compute upper bounds for their depthes. This approximation is obtained as the least model of a system of integer constraints which is generated during control flow analysis.

Let A denote the set of all non–variable subexpressions occurring in program p together with a special symbol $\bullet$ denoting a free variable. For approximating the unfold steps, we use simple constraints on A where restriction to outermost unfolding is taken care of by constraints over **2**. Therefore, our analysis considers the following variables for every occurring subterm t:

$$\begin{array}{ll} (1) & [\![t]\!] : 2^A; \\ (2) & r[\![t]\!] : \mathbf{2}; \\ (3) & d[\![t]\!] : \mathcal{N}; \\ (4) & a[\![t]\!] : \mathcal{N} \end{array}$$

where $[\![t]\!]$ represents a (superset) of the terms obtained from (instances of) t through unfolding; $r[\![t]\!]$ records whether or not (an instance of) t is unfolded at all; $d[\![t]\!]$ gives (an upper bound to) the depth of reduction contexts in which t is called; finally $a[\![t]\!]$ represents (an upper bound to) the depth of occurring instances t.

The algorithm basically consists in iteratively putting up constraints for these variables. Formally, the resulting set S of constraints can be denoted as the least fixpoint

$$S = \mu F$$

where F is a monotone function operating on sets of constraints which commutes with binary "$\cup$". F can be decomposed into

$$F\ x = S_p \cup (\mathrm{New}_p x)$$

[3] Instead of constraint systems, [Se94] considers systems of equations. It is not difficult to see that w.r.t. the minimal model this makes no difference.

S_p is the initial set of constraints, whereas New$_p$ describes how, depending on already found constraints resp. their least model, new constraints are added.

Finally, the set I of integer constraints contained in S are extracted. The least model of these contains the desired information.

```
$[\![x]\!] \supseteq \bullet$,          $x$ free;                                                        (1)
$[\![h\,t_1 \dots t_n]\!] \supseteq h\ t_1 \dots t_n$;                                                   (2)
$d[\![main]\!] \supseteq 1$; if $n > 0$ then $a[\![h\,t_1 \dots t_n]\!] \geq 1 + a[\![t_i]\!]$;          (3)
```

Fig. 3. The set S_p of initial constraints

The definitions of S_p and New_p are contained in Figures 3 and 4.

```
if $[\![main]\!] \supseteq c\ s_1 \dots s_n$                                                              (1)
then $[\![main]\!] \supseteq [\![s_i]\!]$;                                                                (2)
     $r[\![s_i]\!] \supseteq 1$;                                                                          (3)
if $[\![t]\!] \supseteq [\![t']\!]$ and $[\![t']\!] \supseteq h\,t_1 \dots t_n$                           (4)
then $[\![t]\!] \supseteq h\,t_1 \dots t_n$;                                                              (5)
if $r[\![t]\!] \supseteq 1$ then                                                                          (6)
      case $t$ of                                                                                         (7)
      $f\ t_1 \dots t_n$:      $[\![t]\!] \supseteq [\![t^f]\!]$;                                          (8)
                              $[\![x_i^f]\!] \supseteq [\![t_i]\!]$;                                      (9)
                              $r[\![t^f]\!] \supseteq 1$; $d[\![t^f]\!] \geq d[\![t]\!]$; $a[\![x_i^f]\!] \geq a[\![t_i]\!]$;   (10)
      $g\ t_0 \dots t_n$:      $[\![x_i^g]\!] \supseteq t_i$;                                              (11)
                              $r[\![t_0]\!] \supseteq 1$; $a[\![x_i^g]\!] \geq a[\![t_i]\!]$; $d[\![t_0]\!] \geq 1 + d[\![t]\!]$;   (12)
                              if $[\![t_0]\!] \supseteq \bullet$ then for all $c$,                        (13)
                                   $[\![t]\!] \supseteq [\![t^{g,c}]\!]$;                                 (14)
                                   $r[\![t^{g,c}]\!] \supseteq 1$; $d[\![t^{g,c}]\!] \geq d[\![t]\!]$;    (15)
                                   $[\![x_i^{g,c}]\!] \supseteq \bullet$;                                 (16)
                              if $[\![t_0]\!] \supseteq c\ s_{n+1} \dots s_m$                             (17)
                              then $[\![t]\!] \supseteq [\![t^{g,c}]\!]$;                                 (18)
                                   $[\![x_i^{g,c}]\!] \supseteq [\![s_i]\!]$;                             (19)
                                   $r[\![t^{g,c}]\!] \supseteq 1$; $a[\![x_i^{g,c}]\!] \geq a[\![s_i]\!]$; $d[\![t^{g,c}]\!] \geq d[\![t]\!]$;   (20)
      $x$ :                    if $[\![x]\!] \supseteq [\![t']\!]$                                        (21)
                              then $r[\![t']\!] \supseteq 1$; $d[\![t']\!] \geq d[\![t]\!]$;              (22)
      let $x = t_1$ in $t_2$ : $[\![t]\!] \supseteq [\![t_2]\!]$;                                          (23)
                              $[\![x]\!] \supseteq \bullet$;                                              (24)
                              $r[\![t_i]\!] \supseteq 1$; $d[\![t_2]\!] \geq d[\![t]\!]$;                 (25)
                              if $[\![t_1]\!] \supseteq c\ s_1 \dots s_n$                                 (26)
                              then $[\![t_1]\!] \supseteq [\![s_i]\!]$;                                   (27)
                                   $r[\![s_i]\!] \supseteq 1$;                                            (28)
```

Fig. 4. Adding new constraints with New_p

Removing line (3) from the definition of S_p and lines (3), (6), (10), (12), (15), (20), (22), (25) and (28) from the definition of New_p, one obtains an ordinary control flow analysis of p where unfolding may occur in arbitrary positions. Line (1) of S_p means that $\bullet$ is included into $[\![x]\!]$ for all free variables of *main*, whereas line (2) reflects reflexivity of the rewriting relation.

The first two lines in the definition of New_p ensures that deforestation is pushed down outermost constructors. Line (4) simulates transitivity of rewriting. The remaining part of New_p is responsible for correctly approximating unfolding. The most interesting case is when function application t equals $g\ t_0 \dots t_n$ (lines (11) through (19)). Then two subcases may occur. If a constraint $[\![t_0]\!] \supseteq \bullet$ is already present (lines (13) through (16)), we conclude that t_0 possibly may be rewritten to a global variable. Then all right hand sides for g are possible results for t (line (14)). Extra constraints are added to model possible variable bindings. Since t_0 is possibly a free variable all pattern variables receive $\bullet$ as possible value (line (16)). If, on the contrary, a constraint $[\![t_0]\!] \supseteq c\ s_{n+1} \dots s_m$ is found, rewriting with alternative for constructor c is modeled (lines (17) through (19)). Especially, the possible bindings of pattern variables $x_j^{g,c}$ are taken care of through constraints $[\![x_j^{g,c}]\!] \supseteq [\![s_j]\!]$ (line (19)). Finally for the **let** construct, lines (26) and (27) reflect that deforestation is also pushed through outermost constructors of the term the new variable x is bound to, whereas lines (23) and (24) indicate that the value of the whole expression is just the value returned for t_2 where x is treated like a global variable.

Consider, e.g., the following example program of Fig. 5. Unrestricted control flow analysis would result in the set of constraints of Fig. 6 (for simplicity, we only listed constraints of the form $[\![t]\!] \supseteq s$ where $t \neq s$). Observe that from

```
letrec
    fi [] = None
    fi (x:xs) = Any x
    id [] = []
    id (y:ys) = y:(id ys)
in
    fi (id l)
```

Fig. 5. The ‘first’ function `fi`.

$$\begin{array}{rlrl}
[\![\mathtt{l}]\!] \supseteq & \bullet & [\![\mathtt{id\ l}]\!] \supseteq & [\,]\ ,\ \mathtt{y : (id\ ys)} \\
[\![\mathtt{y}]\!] \supseteq & \bullet & [\![\mathtt{ys}]\!] \supseteq & \bullet \\
[\![\mathtt{id\ ys}]\!] \supseteq & [\,]\ ,\ \mathtt{y : (id\ ys)} & [\![\mathtt{fi\ (id\ l)}]\!] \supseteq & \mathtt{None}\ ,\ \mathtt{Any\ x} \\
[\![\mathtt{x}]\!] \supseteq & \bullet & [\![\mathtt{xs}]\!] \supseteq & \mathtt{id\ ys}\ ,\ [\,]\ ,\ \mathtt{y : (id\ ys)}
\end{array}$$

Fig. 6. Unrestricted control flow of the program in Fig. reffirst.

this approximation, one might conclude that `xs` is possibly bound to (some instance of) `y:(id ys)` – which in fact never occurs. By adding the constraints on variables $r[\![t]\!]$, we obtain a control flow analysis of p w.r.t. an outermost reduction strategy. Now, by the conditional in line (6) of new_p, (simulation of) unfolding of t only takes place provided $r[\![t]\!] \supseteq 1$ has already been established. Thus, in our example we obtain the set of constraints of Fig. 7.

$$\begin{array}{rlcrl}
r[\![\texttt{fi (id l)}]\!] & \supseteq 1 & \qquad & r[\![\texttt{id l}]\!] & \supseteq 1 \\
r[\![\texttt{y : (id ys)}]\!] & \supseteq 1 & & [\![\texttt{l}]\!] & \supseteq \bullet \\
[\![\texttt{id l}]\!] & \supseteq [\,]\ ,\ (\texttt{y : (id ys)}) & & [\![\texttt{y}]\!] & \supseteq \bullet \\
[\![\texttt{ys}]\!] & \supseteq \bullet & & [\![\texttt{fi (id l)}]\!] & \supseteq\ ,\ \texttt{None}\ ,\ (\texttt{Any x}) \\
[\![\texttt{x}]\!] & \supseteq \bullet & & [\![\texttt{xs}]\!] & \supseteq \texttt{id ys}
\end{array}$$

Fig. 7. Restricted control flow of the program of Fig. 5.

During the computation of the dataflow constraints on the variables $[\![t]\!]$, integer constraints are generated. Line (3) in the definition of S_p ensures that $a[\![t]\!]$ measures the depth of t – increased by the possible values for variables occurring in t. Lines (10), (12) and (20) are responsible for correctly producing values of these variables whenever these are bound by function unfolding. Accordingly, the constraints on variables $d[\![t]\!]$ iin lines (10), (15), (20) and (25) are responsible for propagating the depth of a reduction context of a function call or **let** statement to the residual subexpression after unfolding. Additionally, in line (12), the depth is increased when passing from a call to a g–function to its 0-th argument. Finally in line (22), the depth of a reduction context in which a variable occurs is propagated to the term to which it is possibly bound. The set of integer constraints generated for the program in Fig. 5 is given by Fig. 8. Clearly, since the original program was rather trivial, this system is trivial as well with least model mapping, e.g., $a[\![\texttt{xs}]\!]$ and $d[\![\texttt{id l}]\!]$ to 1. However, we have:

$$\begin{array}{rlcrl}
d[\![\texttt{id l}]\!] & \geq 1 + d[\![\texttt{fi (id l)}]\!] & \qquad & a[\![\texttt{fi (id l)}]\!] & \geq 1 + a[\![\texttt{id l}]\!] \\
a[\![\texttt{id l}]\!] & \geq 1 + a[\![\texttt{l}]\!] & & a[\![\texttt{id ys}]\!] & \geq 1 + a[\![\texttt{ys}]\!] \\
a[\![\texttt{x}]\!] & \geq a[\![\texttt{y}]\!] & & a[\![\texttt{xs}]\!] & \geq a[\![\texttt{id ys}]\!]
\end{array}$$

Fig. 8. The set of generated integer constraints

Theorem 1. *Consider program* $p \equiv$ **letrec** *defs* **in** *main, set* S *of constraints generated by the analysis, and set* I *of integer constraints contained in* S *with least model* μI.

Safety:
Assume during deforestation of p, $\mathcal{D}$ is applied to $e[t]$ for function application t. Then

1. $\mu I\ d[\![t]\!] \geq \mathrm{depth}(e)$;
2. *If* $t \equiv f\ t_1 \ldots t_n$ *then* $\mu I\ a[\![x_i^f]\!] \geq \mathrm{depth}(t_i)$ *for all* i.
   If $t \equiv g\ (c\ t_{n+1} \ldots t_m)\ t_1 \ldots t_n$ *then* $\mu I\ a[\![x_i^g]\!] \geq \mathrm{depth}(t_i)$ *for* $i = 1, \ldots, n$, *and* $\mu I\ a[\![x_i^{g,c}]\!] \geq \mathrm{depth}(t_i)$ *for* $i = n+1, \ldots, m$.

Efficiency:
S together with μI can be computed in polynomial time.

Proof. The safety part follows from Prop. 4 of the next section. The fast algorithm to compute S essentially consists (of an adaptation) of Heintze's algorithm for computing a normalized system of set constraints in [He94] where additionally the constraints for variables $r[\![\ldots]\!]$ are considered and the integer constraints in I are generated. Note that, theoretically Heintze's algorithm has cubic complexity. In praxis, however, we found that it behaves quite well on all example programs. Finally for the computation of μI, the fast algorithm of [Se94] can be employed which has runtime linear in the size of I. □

Note that μI indeed contains all information necessary to determine whether or not $\mathcal{D}$ stops. We obtain as a corollary:

Corollary 2. *If $\mathcal{D}$ encounters an infinite number of expressions $e[t]$ then either (1) or (2) holds:*

(1) Some function application t exists with $\mu I\ d[\![t]\!] = \infty$;
(2) Some formal x exists with $\mu I\ d[\![x]\!] = \infty$. □

5 Correctness

In order to prove safety for our analysis, we put up a reduction relation "$\Rightarrow$" between the representation of terms chosen as input to $\mathcal{D}$ such that $t \overset{*}{\Rightarrow} t'$ iff of t calls deforestation of t'. Thus, this reduction relation essentially formalizes outermost unfolding of terms possibly containing free variables. Here are the reduction rules defining "$\Rightarrow$":

1. $e[f\ t_1 \ldots t_n] \Rightarrow t^f[(t_i, E)/x_i^f]$
2. $e[g\ (c\ t_{n+1} \ldots t_m)\ t_1 \ldots t_n] \Rightarrow e[t^{g,c}\ [t_i/x_i^{g,c}, t_j/x_i^g]]$
3. $e[g\ x\ t_1 \ldots t_n] \Rightarrow e[t^{g,c}[(t_j/x_i^g]$ for all c if $x \notin \mathrm{dom}(E)$;
4. $c\ t_1 \ldots t_n \Rightarrow t_i$ for all i;
5. $e[\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2] \Rightarrow e[t_2]$ and finally,
6. $e[\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2] \Rightarrow t_1$.

Instead of abstracting this reduction system we prefer to abstract the corresponding system where substitutions are made explicit. This allows us, for instance, to keep track of instances of subexpressions. Therefore instead of terms we consider stacks of pairs consisting of expressions and environments. Environments E are used for making substitutions explicit whereas stacks σ are introduced to model the outermost unfolding streategy. The stack is meant to store suspended function applications. We have

$$\begin{aligned} E &::= \emptyset \mid [(t,E)/x] : E' \\ \sigma &::= \epsilon \mid (t,E)\ \sigma \end{aligned}$$

Here is the way how to determine the term denoted by a pair of expression and environment resp. a stack of such pairs.

- $u\llbracket t, E \rrbracket = t$ if t contains no variables from $\mathrm{dom}(E)$.
- $u\llbracket x, E \rrbracket = \llbracket Ex \rrbracket$ if $x \in \mathrm{dom}(E)$.
- $u\llbracket h\ t_1 \ldots t_n, E \rrbracket = h\ u\llbracket t_1, E \rrbracket \ldots u\llbracket t_n, E \rrbracket$.
- $u\llbracket \textbf{let}\ x = t_1\ \textbf{in}\ t_2, E \rrbracket = \textbf{let}\ x = u\llbracket t_1, E \rrbracket\ \textbf{in}\ \llbracket t_2, E \rrbracket$.
- $u\llbracket (t', E')(g\ t_0, t_1 \ldots t_n, E)\ \sigma \rrbracket = u\llbracket (g\ v\ t_1, \ldots, t_n, E)\ \sigma \rrbracket$ where $v = u\llbracket t', E' \rrbracket$.

On such stacks we introduce a reduction relation "$\rightarrow$" which is going to simulate our reduction relation "$\Rightarrow$" on terms. We define:

1. $(x, E)\ \sigma \rightarrow (Ex)\ \sigma$ if $x \in \mathrm{dom}(E)$;
2. $(f\ t_1 \ldots t_n, E)\ \sigma \rightarrow (t^f, [(t_i, E)/x_i^f])\ \sigma$;
3. $(g\ t_0 \ldots t_n, E)\ \sigma \rightarrow (t_0, E)(g\ t_0 \ldots t_n, E)\ \sigma$;
4. $(c\ t_{n+1} \ldots t_m, E')(g\ t_0 \ldots t_n, E)\ \sigma \rightarrow (t^{g,c}, [(t_i, E')/x_i^{g,c}, (t_j, E)/x_i^g])\ \sigma$;
5. $(x, E')(g\ t_0 \ldots t_n, E)\ \sigma \rightarrow (t^{g,c}, [(t_j, E)/x_i^g])\ \sigma$ for all c if $x \notin \mathrm{dom}(E)$;
6. $(c\ t_1 \ldots t_n, E) \rightarrow (t_i, E)$ for all i;
7. $(\textbf{let}\ x = t_1\ \textbf{in}\ t_2, E)\ \sigma \rightarrow (t_2, E)\ \sigma$ and finally,
8. $(\textbf{let}\ x = t_1\ \textbf{in}\ t_2, E)\ \sigma \rightarrow (t_1, E)$.

By induction on the length of reductions we verify:

Proposition 3. *Assume $t = u\llbracket \sigma \rrbracket$. Then*

1. *$t \overset{*}{\Rightarrow} t'$ implies $\sigma \overset{*}{\rightarrow} \sigma'$ for some σ' with $u\llbracket \sigma' \rrbracket = t'$; and*
2. *$\sigma \overset{*}{\rightarrow} \sigma'$ implies $t \overset{*}{\Rightarrow} t'$ for $t' = u\llbracket \sigma' \rrbracket$.* □

By Prop. 3, it suffices to abstract relation "$\rightarrow$" on stacks. For stack σ, let $\alpha\sigma$ denote the following set of simple constraints.

- If $\sigma = \epsilon$ then $\alpha\sigma = \emptyset$;
- If $\sigma = (t, E)$ then $\alpha\sigma = \{\llbracket \mathit{main} \rrbracket \supseteq t\} \cup \alpha E$;
- If $\sigma = (t, E)(g\ t_0 \ldots t_n)\ \sigma$ then $\alpha\sigma = \{\llbracket t_0 \rrbracket \supseteq t\} \cup \alpha E \cup \alpha((g\ t_0 \ldots t_n)\ \sigma)$

where αE is defined by:

- If $E = \emptyset$ then $\alpha E = \emptyset$;

- If $E = (t, E_1)/x) : E_2$ then $\alpha E = \{[\![x]\!] \supseteq t\} \cup \alpha E_1 \cup \alpha E_2$.

By induction on the lengthes of reduction paths we prove:

Proposition 4. *Let I denote the set of integer constraints contained in S, and μI its least model. Assume $(main, \emptyset) \xrightarrow{*} (t, E)\ \sigma$. Then*

1. *$\alpha((t, E)\ \sigma) \subseteq S$;*
2. *$r[\![t]\!] \subseteq 1$ is in S;*
3. *$\mu I\ d[\![t]\!] \geq |\sigma|$;*
4. *$\mu I\ a[\![t]\!] \geq \mathrm{depth}(u[\![t, E]\!])$;*
5. *$\mu I\ a[\![x]\!] \geq \mathrm{depth}(u[\![x, E]\!])$ for every $x \in \mathrm{dom}(E)$.*

If furthermore $(t, E)\ \sigma \xrightarrow{*} (t', E')\ \sigma$ is a reduction path where all intermediate stacks are of the form $\sigma_1\sigma$, then

6. *$[\![t]\!] \supseteq t'$ is in S as well.* □

This finishes the proof of statement 1 of Theorem 1.

6 Characterizing Dangerous Subexpressions

Assume we found that deforestation may not stop when applied to certain subexpressions. Then the next task is to spot more precisely the sources of nontermination. Therefore in this section, we take a closer look at the set I of integer constraints generated by the analysis. I has the following two properties:

1. The subsystems I_d and I_a containing variables $d[\![t]\!]$ resp. $a[\![t]\!]$ are disjoint;
2. The right hand sides are of one of the two forms: $X \geq Y$ or $X \geq 1 + Y$ for variables X and Y.

In [Se94], a characterization is given of all X with $\mu I\ X = \infty$. It is given in terms of the dependence graph of the system. Let $I' = I_a$ (the case $I' = I_d$ is completely analogous). The *dependence graph* for I' is the directed graph G whose set of nodes equals the set of variables of I', and whose set of edges consists of all (X, Y) with $Y \geq X$ or $Y \geq 1 + X$ occurring in I'. Specializing the characterization of [Se94] to our application we find:

Proposition 5. *The set J of all t with $\mu I'\ a[\![t]\!] = \infty$ is the smallest set (w.r.t. set inclusion) containing all t such that*

(1) $a[\![t]\!]$ is contained in a strong component of G which also contains variables $a[\![t_1]\!], a[\![t_2]\!]$ for some constraint $a[\![t_1]\!] \geq 1 + a[\![t_2]\!]$ is in I'.

(2) $a[\![t]\!]$ is reachable in G from some $a[\![t']\!]$ with $t' \in J$. □

Example 3. As an example, consider again the reverse function with accumulating parameter from the introduction. In this case, (among others) the following integer contraints on variables $a[\![\ldots]\!]$ are generated:

$$\begin{array}{cc} a[\![\mathtt{xs}]\!] \geq a[\![\mathtt{l}]\!] & a[\![\mathtt{y}]\!] \geq a[\![\mathtt{z:y}]\!] \\ a[\![\mathtt{z:y}]\!] \geq 1 + a[\![\mathtt{z}]\!] & a[\![\mathtt{z:y}]\!] \geq 1 + a[\![\mathtt{y}]\!] \end{array}$$

In deed, the least model maps $a[\![\mathtt{l}]\!]$, $a[\![\mathtt{xs}]\!]$, and $a[\![\mathtt{z}]\!]$ to 0, and $a[\![\mathtt{z:y}]\!]$ and $a[\![\mathtt{y}]\!]$ to ∞.

In general, taking into account the way integer contraints are generated by New_p we obtain:

Corollary 6. *1. Assume that for no program variable x, $a[\![x]\!]$ is contained in a strong component of G which also contains variables $a[\![t_1]\!], a[\![t_2]\!]$ for some constraint $a[\![t_1]\!] \geq 1 + a[\![t_2]\!]$ in I_a. Then $\mu I_a\ a[\![x]\!] < \infty$ for all x.*

2. Assume that for no function application $t = h\ t_1 \ldots t_n$, $d[\![t]\!]$ is contained in a strong component of G which also contains variables $d[\![t_1]\!], d[\![t_2]\!]$ for some constraint $d[\![t_1]\!] \geq 1 + d[\![t_2]\!]$ in I_d. Then $\mu I_d\ d[\![x]\!] < \infty$ for all x. □

Remark that items (1) and (2) of Cor. 6 are the translations of Sørensen's criteria for accumulating parameters resp. obstructing function calls [Sø94b]. Our characterization is based on simple graph properties. It follows that the set of dangerous variables resp. function calls can be computed efficiently, namely in time linear in I. Moreover, we can be more specific. One problem with deforestation is that the transformed program may be very large. Therefore instead of computing the set of variables mapped to ∞, one may compute μI (which is possible in linear time as well) and then determine those variables $a[\![t]\!]$ resp. $d[\![t]\!]$ whose values exceed a certain threshold c.

7 Discussion and Conclusion

In this paper we considered a termination analysis for deforestion. In essence, it boils down to a control flow analysis for a CBN rewrite system which is used to produce a system of integer constraints. Their least model contains the information about possible depthes of reduction contexts resp. arguments we are aiming at. The advantages of this approach are that it allows for a clear exposition of the analysis, a simple correctness proof together with a fast implementation.

Further investigations should include a generalization to higher order deforestation and applications of these ideas to other program analysis problems. One of these might be, e.g., finite topology detection of concurrent functional languages [RNN94].

8 References

[A86] N. Andersen: Approximating Term Rewrite Systems by Regular Tree Grammars. Report Nr. 86/16, Data Logisk Institut, Københavns Universitet, 1986

[Ch90] Wei–Ngan Chin: Automatic Methods for Program Transformation. Ph.D. Thesis, Imperial College, University of London, July 1990

[Ch94] Wei–Ngan Chin: Safe Fusion of Functional Expressions II: Further Improvements. F. Functional Programming. 1994

[Co93] C. Consel: A Tour of Schism: A Partial Evaluation System for Higher–Order Applicative Languages. Proc. PEPM'93, 2nd ACM SIGPLAN Symp. on Partial Evaluation and Semantics–Based Program Manipulation, 145–154, 1993

[CC95] P. Cousot, R. Cousot: Compositional and Inductive Semantic Definitions in Fixpoint, Equational, Constraint, Closure-condition, Rule-based and Game-theoretic Form. Proc. CAV'95, LNCS 939, 293–308, 1995

[De90] A. Deutsch: On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher–Order Functional Specifications. Proc. 17th POPL, 157–168, 1990

[FW88] A. Ferguson, P. Wadler: When will deforestation stop. In: 1988 Glasgow Workshop on Functional Programming, Research Rep. 89/R4, 1988

[He94] N. Heintze: Set–based Analysis of ML Programs. Proc. of the 1994 ACM Conference on Lisp and Functional Programming, 306–317, Lisp Pointers, Orlando, Florida, June 1994

[JM79] N.D. Jones, S.S. Muchnick: Flow Analysis of LISP–like Structures. Proc. 6th ACM Symp. on Principles of Programming Languages, 244–256, 1979

[J87] N.D. Jones: Flow Analysis of Lazy Higher–Order Functional Programs. In: S. Abramsky, C. Hankin (eds): *Abstract Interpretation.* Ellis Horwood, 103–122, 1987

[RNN94] H. Riis Nielson, F. Nielson: Static and Dynamic Processor Allocation for Higher Order Concurrent Languages. Tech. Report DAIMI PB483, Dept. of Computer Science, Aarhus, 1994; short version in: Proc. TAPSOFT'95

[Pa94] J. Palsberg: Closure Analysis in Constraint Form. ACM Transactions on Programming Languages 17, 47–82, 1995 (preliminary version in CAAP'94)

[PO95] J. Palsberg, P. O'Keefe: A Type System Equivalent to Flow Analysis. To appear in: ACM Transactions on Programming Languages and Systems, 1995

[PS95] J. Palsberg, M.I. Schwarzbach: Safety Analysis versus Type Inference. Inf. and Computation 118, 128–141, 1995

[R69] J. C. Reynolds: Automatic Computation of Data Set Definitions. Information Processing, Vol. 68, 456–461, 1969

[Sa95] D. Sands: Proving Correctness of Recursion–Based Automatic Program Transformations. Proc. TAPSOFT'95, LNCS, 681–695, 1995

[Se94] H. Seidl: Least Solutions of Equations over $\mathcal{N}$. Proc. ICALP'94, LNCS 820, 400–411, 1994

[Sh91] O. Shivers. Data–Flow Analysis and Type Recovery in Scheme. In: Peter Lee (ed.): Topics in Advanced Language Implementation. MIT Press, 47–87, 1991

[Sø94a] Turchin's Supercompiler Revisited. Master's Thesis. DIKU, University of Copenhagen, March 1994

[Sø94b] M.H. Sørensen: A Grammar–based Data–flow Analysis to Stop Deforestation. Proc. CAAP'94, LNCS 787, 335–351, 1994

[SGJ94] M.H. Sørensen, R. Glück, N.D. Jones: Towards Unifying Partial Evaluation, Deforestation, Supercompilation and GPC. Proc. ESOP'94, LNCS 788, 485–500, 1994

[Wa88] P. Wadler: Deforestation: Transforming Programs to Eliminate Trees. TCS 73, 231–248, 1990; (preliminary version in Proc. ESOP'88, LNCS 300)

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