Constraints to Stop Deforestation

H. Seidl

*FB IV - Informatik, Universität Trier,
D-54286 Trier, Germany.* seidl@uni-trier.de

M.H. Sørensen

*Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark.* rambo@diku.dk

Abstract

Wadler's deforestation algorithm eliminates intermediate data structures from functional programs. To be suitable for inclusion in a compiler, deforestation must terminate on all programs. Several techniques exist to ensure termination of deforestation on all first-order programs, but general techniques for higher-order programs were introduced only recently by first Hamilton and then Marlow.

We present a new technique for ensuring termination of deforestation on all higher-order programs that allows useful transformation steps prohibited in Hamilton's and Marlow's techniques. The technique uses a constraint-based higher-order control-flow analysis.

We also relate our technique to previous approaches to termination of first- and higher-order deforestation in some detail.

1 Introduction

Lazy, higher-order, functional programming languages lend themselves to an elegant style of programming which uses intermediate data structures [31]. However, this style also leads to inefficient programs.

Example 1 Consider the following program:

$$\begin{array}{l} \textbf{letrec} \\ \ a = \lambda xs, ys.\, \textbf{case}\ xs\ \textbf{of}\ [\,] \to ys;\ (t : ts) \to t : a\ ts\ ys \\ \textbf{in}\ \lambda us, vs, ws.\ a\ (a\ us\ vs)\ ws \end{array}$$

Preprint submitted to Elsevier Science *1 April 1997*

Program syntax is introduced officially in Section 2; here we introduce some terminology useful for the informal introduction in the present section. The term $\lambda us, vs, ws.\ a\ (a\ us\ vs)\ ws$ is the *main* term, and $a = \cdots$ is a *definition* with *right hand side* $\lambda xs, ys.$ **case** xs **of** $[] \rightarrow ys;\ (t : ts) \rightarrow t : a\ ts\ ys$. In this *case-expression,* xs is the *selector*, and ys and $t : a\ ts\ ys$ are the *branches.* In the *call* $a\,ts\,ys$, ts and ys are *arguments*. We assume that the reader is familiar with the notions of *free* and *bound* variables, and the usual conventions to distinguish free and bound (as well as distinct bound) variables. A variable which occurs bound more than once in the main term, or in the right hand side of a definition, is *non-linear* (in a case-expression we count only the occurrences in the branch with the most occurrences). Although the program contains λ-abstractions, it is a first-order program: abstractions are only used for the formal parameters of a, and all calls to a have exactly two arguments.

The term $\lambda us, vs, ws.\ a\ (a\ us\ vs)\ ws$ appends the three lists us, vs, and ws. Appending us and vs results in an intermediate list to which ws is appended. Allocation and deallocation of the intermediate list at run-time is expensive. Sacrificing clarity for efficiency, we would prefer a program like the following.

$$
\begin{array}{l}
\textbf{letrec} \\
\quad da = \lambda xs, ys, zs.\ \textbf{case}\ xs\ \textbf{of}\ [] \rightarrow a\ ys\ zs;\ (t : ts) \rightarrow t : da\ ts\ ys\ zs \\
\quad a \ \ = \lambda ys, zs.\ \textbf{case}\ ys\ \textbf{of}\ [] \rightarrow zs;\ (t : ts) \rightarrow t : a\ ts\ zs \\
\textbf{in}\ \lambda us, vs, ws.\ \ da\ us\ vs\ ws
\end{array}
$$

In Mark Jones' *Gofer*, the first program uses approximately 13 percent more time and 7 percent more space to append three constant lists of equal length. □

Ideally we should enjoy the benefits of both elegance and efficiency by writing the first version and have it translated to the second automatically, e.g., by our compiler. Some early techniques for this are due to Burstall and Darlington [6], Manna and Waldinger [35], Darlington [14], Kierburtz and Schultis [32], Feather [15], Turchin [62], Bird [3], Wadler [63–65], and Scherlis [48].

This paper is about Wadler's *deforestation* [17,66,68], an algorithm eliminating intermediate data structures from first-order functional programs in which

(i) no definition contains an argument which is not a variable;
(ii) no definition contains a selector which is not a variable;
(iii) no definition nor the main term contains a non-linear variable.

We call a program *treeless* if it satisfies (i-ii), and *linear* if it satisfies (iii). In a treeless program the right hand sides do not construct intermediate data structures; this property guarantees termination of deforestation. In a linear program no duplication occurs; this ensures that the new program is at least as efficient as the original. For instance, the first program in Example 1 is treeless and linear, and can be translated to the second automatically by deforestation.

Wadler also introduced *blazed* deforestation, a variant where the input program's main term and right hand sides have certain marks, and where the algorithm skips over marked subterms. The act of putting such marks on a program is *blazing.* The problem remains to blaze each program so that

(i) blazed deforestation of the blazed program terminates;
(ii) the resulting program is no less efficient than the original one.

A technique for blazing is *termination-safe* and *efficiency-safe* if it satisfies (i) and (ii), respectively, for all programs, and *safe* if it satisfies both (i) and (ii) for all programs. Since transformation skips over marked subterms, intermediate data structures produced by such subterms will not be removed. Thus, as few subterms as possible should receive marks, but enough subterms should receive marks to ensure safety.

Wadler blazes programs on the basis of *types.* For instance, a subterm of integer type does not produce as a result a data structure, so nothing is lost by marking the subterm. The type-based blazing is not generally safe (and was not mean to be).

Chin [7,8,10,11] safely blazes all first-order programs by, roughly, marking all subterms violating the linear, treeless syntax:

(i) for every definition, all arguments that are not variables;
(ii) for every definition, all selectors that are not variables;
(iii) for every definition and the main term, all non-linear variables.

Here (i-ii) and (iii) account for termination-safety and efficiency-safety, respectively. Chin also presents many extensions of this blazing scheme. For instance, he refines (i) by not requiring that arguments of a non-recursive function be marked. He also notes that the syntactic non-linearity condition (iii) might be replaced by a semantic condition, stating that all terms evaluated more than once should be marked. More extensions were devised by Chin and Khoo [13].

Hamilton and Jones [25,26] use static analyses to blaze first-order programs, but in some cases blazed deforestation loops indefinitely on the blazed program. Later, Hamilton [22] describes a safe blazing scheme similar to Chin's (i-iii). In his thesis [21] he gives another safe blazing scheme, replacing (ii-iii) by similar semantic conditions, roughly:

(ii) all terms appearing as a case-selector after a number of evaluation steps;
(iii) all terms that will be evaluated more than once.

Both of (ii-iii) are approximated by a *usage counting* analysis.

The safe blazing schemes by Chin and Hamilton are—at least partly—syntactic: they mark parts of the program that violate some version of the linear, treeless syntax, thereby failing to improve such subterms. In contrast, Sørensen and Seidl [59,50] compute a constraint-based control-flow analysis which approximates whether deforestation of a given program terminates. This information is used to blaze in a termination-safe way. The technique is conservative over Wadler's technique (and the core of Chin's and Hamilton's syntactic techniques) in the sense that for any treeless program the technique discovers that no marks are required. Moreover, for some non-treeless programs, it discovers that no marks are necessary.

What has been said so far concerns *first-order* programs. However, languages like Haskell and Miranda include higher-order functions which should be transformed too. Along with the above mentioned techniques to ensure termination of first-order deforestation came some attempts to reduce the problem of ensuring termination of higher-order deforestation to the first-order case.

Wadler [68] considers programs with *higher-order macros*. Any such program typable in the Hindley-Milner [29,38] type system can be expanded out to a first-order program, and transformed with first-order deforestation. These programs include applications of the *fold* and *map* functions, but exclude useful constructions, e.g., lists of functions.

Chin [7,8,10,11] starts out with a higher-order program and uses higher-order removal [7,9,12] to eliminate some higher-order parts, resulting in a program in a restricted higher-order form. He then adopts a version of deforestation applicable to blazed programs in the restricted higher-order form, and marks remaining higher-order parts, as well as first-order parts violating the treeless syntax. While deforesting such a program, higher-order subterms may reappear, and these are removed by the higher-order removal algorithm along the way. The whole process terminates if the program is typable in the Hindley-Milner type system, but a more efficient and transparent approach is desirable.

The first formulation of deforestation applicable directly to higher-order programs is due to Marlow and Wadler [36], who leave open the question of guaranteeing termination. Sands [44,46] and Nielsen and Sørensen [40] give other formulations of higher-order deforestation, but are concerned with other problems than ensuring termination.

The first direct solution to the termination problem for higher-order deforestation is due to Hamilton [24], who presents a blazed higher-order deforestation algorithm and introduces a notion of higher-order linear treelessness:

(i) no definition contains an application with a non-variable argument;

(ii) no definition contains a case-expression with a non-variable case-selector;
(iii) no definition nor the main term contains a non-linear variable.

For first-order programs (written in the higher-order style of Example 1) this is equivalent to first-order linear treelessness. Hamilton's main result states that blazed higher-order deforestation of any Hindley-Milner typable program terminates with a program which is no less efficient than the original, if all parts of the program violating the higher-order treeless syntax are marked.

Hamilton's work inspired Marlow to re-evaluate his earlier research [37]. He had also discovered a version of blazed higher-order deforestation, similar to Hamilton's formulation, and now gave a very similar notion of higher-order treelessness and a proof that transformation of any Hindley-Milner typable higher-order program terminates, if all parts of the program violating the higher-order treeless syntax are marked. Marlow has implemented the technique in the Glasgow Haskell compiler, and reports substantial experiments.

The higher-order treeless syntax requires arguments of applications and selectors of case-expressions to be variables. This entails marking and thereby skipping over parts of programs that could have been improved.

Example 2 Consider the following program.

$$
\begin{array}{l}
\textbf{letrec} \\
\quad \begin{array}{lcl}
c & = & \lambda x, xs.\, x : xs \\
\mathit{foldr} & = & \lambda f, a, l.\, \textbf{case}\ l\ \textbf{of}\ [\,] \to a;\ (z : zs) \to f\ z\ (\mathit{foldr}\ f\ a\ zs)
\end{array} \\
\textbf{in}\ \lambda us, vs, ws.\, \mathit{foldr}\ c\ ws\ (\mathit{foldr}\ c\ vs\ us)
\end{array}
$$

The term *foldr c vs us* is higher-order—it uses functions as arguments. The whole program is equivalent to the program in Example 1, and we would expect to be able to transform it into the efficient program in Example 1. This is indeed what happens when we deforest the program. However, the techniques by Hamilton and Marlow require that the argument *foldr f a zs* in the definition of *foldr* be marked, and this prevents the desired transformation. □

There are many such examples. Chin [11] shows that some shortcomings of the treeless syntax can be avoided by ad-hoc extensions of deforestation. The necessity of such extensions stems from the fact that the blazing scheme is syntactic; it does not take into account what happens during deforestation. In contrast, Seidl and Sørensen [52] give a single technique coping with many of these problems by generalizing their constraint-based control-flow analysis [59,50] to work for higher-order deforestation. The technique is partly conservative over the ones based on the higher-order treeless syntax: for a certain large class of higher-order treeless programs the technique discovers that no marks are required. Moreover, for some non-treeless programs, it discovers that no marks are necessary. The rest of this paper gives a fuller presentation of that technique.

Section 2 describes a simple higher-order language and its first-order fragment. Section 3 presents blazed higher-order deforestation. Section 4 describes the archetypes of the termination problems in first- and higher-order deforestation. Section 5 introduces constraints, and Section 6 presents the constraint-based analysis for approximation of deforestation. Section 7 shows that the analysis is correct, and Section 8 uses it to ensure termination of deforestation. Section 9 is concerned with conservativity over methods based on the higher-order treeless syntax. Section 10 discusses related work, not mentioned above.

2 Language and notation

In this section we describe the higher-order language with which we shall be concerned throughout the paper, and its first-order fragment.

Definition 3 (Higher-order language) Let c, x, and f range over names for *constructors*, *variables*, and *functions*, respectively. Let t, q, d, and p range over *terms*, *patterns*, *definitions*, and *programs*, respectively, as defined by the following grammar.

$$t ::= x \mid c\, t_1 \dots t_n \mid \textbf{case}\ t_0\ \textbf{of}\ \ q_1 \rightarrow t_1; \dots ; q_k \rightarrow t_k \mid \lambda x.\, t \mid t\, t' \mid f \mid \textbf{let}\ x{=}t\ \textbf{in}\ t'$$

$$q ::= c\, x_1 \dots x_n$$

$$d ::= f = t$$

$$p ::= \textbf{letrec}\ d_1; \dots ; d_n\ \textbf{in}\ t$$

(where $n \geq 0, k > 0$). The t_0 in case-expressions is the *selector*, The x_i in a pattern are *pattern variables*. In applications t is the *operator* and t' the *argument*. In definitions, f is the *left hand side* and t is the *right hand side*. In a let-expression $\textbf{let}\, x{=}t\, \textbf{in}\, t'$, x may not occur in t. Constructors have fixed arity. No variable may occur more than once in a pattern. No two patterns in a case expression may contain the same constructor. For every function call a in a right hand side, there must be a definition with left hand side a. Programs must be *closed*, i.e., all variables of t in definitions $f = t$ and in programs $\textbf{letrec}\, d_1; \dots ; d_n\, \textbf{in}\, t$ must be bound. We assume that terms of form $\textbf{case}\ (\lambda x.\, t)\ \textbf{of}\ \ q_1 \rightarrow t_1; \dots ; q_n \rightarrow t_n$ and $(c\, t_1 \dots t_n)\, t$ never arise. We also assume that no case-expressions of form $\textbf{case}\ (c\, t'_1 \dots t'_m)\, \textbf{of}\ \ q_1 \rightarrow t_1; \dots ; q_k \rightarrow t_k$ arise, where no q_i has constructor c, or where some q_i is $c\, x_1 \dots x_k$ and $m \neq k$. $\mathrm{FV}(t)$ denotes the set of free variables in t. We identify terms differing only in names for bound variables, and adopt the usual conventions to avoid confusion between free and bound variables and between different bound variables. We also use the usual conventions for association of parentheses. We write $\lambda x_1, \ldots, x_n. t$ for $\lambda x_1. \ldots \lambda x_n. t$. The list constructors *Cons* and *Nil* are written : and []. Instead of $s_1 : \ldots : s_n : []$ we also write $[s_1, \ldots, s_n]$. Substitution of t' for x in t is written $t\{x := t'\}$. □

The semantics of the language is call-by-need [1]. We do not assume that programs are typed.

As in [36], the let-construct is used instead of the marks mentioned in Section 1. Instead of marking parts of a program and letting deforestation skip over marked subterms, we transform these parts of the program into let-expressions and let deforestation skip over let-expressions. This yields less syntactic overhead than working with marks.

We shall occasionally be concerned with the first-order fragment of our language, defined next. Every first-order term and program is also a higher-order term and program. On the other hand, λ-abstraction and application is used in the first-order language only to allow functions to have arguments.

Definition 4 (First-order language) Let c, x, and f range over constructor names, variable names, and function names, respectively. Let t, q, d, and p range over *first-order* terms, patterns, definitions, and programs, respectively, as defined by:

$$
\begin{aligned}
t &::= x \mid c\, t_1 \ldots t_n \mid \textbf{case}\ t\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_k \to t_k \mid f\ t_1 \ldots t_n \mid \textbf{let}\ x{=}t\ \textbf{in}\ t' \\
q &::= c\, x_1 \ldots x_n \\
d &::= f = \lambda x_1, \ldots, x_n. t \\
p &::= \textbf{letrec}\, d_1; \ldots; d_n\ \textbf{in}\ \lambda x_1, \ldots, x_m. t
\end{aligned}
$$

(where $m, n \geq 0, k > 0$). The same restrictions apply as in the preceding definition. In addition, all function calls must be saturated; that is, if one of the definitions in a program is $f = \lambda x_1, \ldots, x_n.\ t$, and the term $f\ t_1 \ldots t_m$ occurs somewhere in the program, then we require $n = m$. □

Definition 5 For a term, a definition, or a program u, define the size $|u|$, by:

$$
\begin{aligned}
&|x| = |f| && = 1\\
&|c\, t_1 \ldots t_n| && = 1 + |t_1| + \ldots + |t_n|\\
&|\mathbf{case}\ t_0\ \mathbf{of}\ \ q_1 \rightarrow t_1; \ldots; q_k \rightarrow t_k| && = 1 + |t_0| + |q_1| + |t_1| + \ldots |q_k| + |t_k|\\
&|\lambda x.\, t| = |f = t| && = 1 + |t|\\
&|t\, t'| = |\mathbf{let}\ x{=}t\ \mathbf{in}\ t'| && = 1 + |t| + |t'|\\
&|\mathbf{letrec}\, d_1; \ldots; d_n\ \mathbf{in}\ t| && = |t| + |d_1| + \ldots |d_n|
\end{aligned}
$$

3 The higher-order deforestation algorithm

In this section we present higher-order blazed deforestation. We shall do so by giving rules for transforming terms. For this, we need some notation to select, e.g., a function call in a term and replace the call by the body of the function. The deforestation algorithm is a generalization of a call-by-name interpreter to terms with free variables, so there is always a unique subterm whose reduction is forced. For instance, to find out which branch to choose in

$$\mathbf{case}\ f\ t\ \mathbf{of}\ \ [\,] \rightarrow [\,];\ (x : xs) \rightarrow x : a\ xs\ ys$$

we are forced to unfold the call to f. The forced call f is the *redex* and the surrounding part of the term, i.e.,

$$\mathbf{case}\ \langle\rangle\ t\ \mathbf{of}\ \ [\,] \rightarrow [\,];\ (x : xs) \rightarrow x : a\ xs\ ys$$

is the *context*.

Definition 6 Let e, r, o range over *contexts*, *redexes*, and *observables*, respectively, as defined by the grammar:

$$
\begin{aligned}
e &::= \langle\rangle \mid \mathbf{case}\ e\ \mathbf{of}\ \ q_1 \rightarrow t_1; \ldots; q_n \rightarrow t_n \mid e\, t\\
r &::= \mathbf{let}\ x{=}t\ \mathbf{in}\ t' \mid (\lambda x.\, t)\ t' \mid f \mid \mathbf{case}\ (c\, t_1 \ldots t_n)\ \mathbf{of}\ \ q_1 \rightarrow s_1; \ldots; q_k \rightarrow s_k \mid\\
&\qquad \mathbf{case}\ (x\ t_1 \ldots t_n)\ \mathbf{of}\ \ q_1 \rightarrow s_1; \ldots; q_k \rightarrow s_k\\
o &::= c\, t_1 \ldots t_n \mid x\, t_1 \ldots t_n \mid \lambda x.\, t
\end{aligned}
$$

(The term $x\, t_1 \ldots t_n$ is simply a nested application.) Let $e\langle t\rangle$ denote the result of replacing $\langle\rangle$ in e by t. $\square$

Every term t is either an observable or decomposes uniquely into a context e and redex r with $t \equiv e\langle r\rangle$, as is easy to prove by induction on t. Stating that $t \equiv e\langle r\rangle$ does not mean that t has any brackets "$\langle$" and "$\rangle$". On the contrary, t is a term. However, e is a context containing an occurrence of $\langle\rangle$, and replacing this occurrence of $\langle\rangle$ by r yields t. This provides a way of finding the next subterm to reduce in a term: in every step deforestation decomposes t into e and r such that $t \equiv e\langle r\rangle$, unfolds r one step yielding r', and continues with $e\langle r'\rangle$.

The following clauses of the deforestation algorithm $[\![.]\!]$ are mutually exclusive and together exhaustive. The definition of $[\![.]\!]$ is inspired by Hamilton [23], but the use of let-expressions rather than marks, and of explicit contexts, has reduced the number of rules from 16 to 8.

Definition 7 (Blazed Deforestation) Define $[\![.]\!]$ on programs and terms:

$$
\begin{array}{lll}
(0) & [\![\mathbf{letrec}\, d_1 \ldots d_n \,\mathbf{in}\, t]\!] & = [\![t]\!] \\
(1) & [\![x\, t_1 \ldots t_n]\!] & = x\, [\![t_1]\!] \ldots [\![t_n]\!] \\
(2) & [\![c\, t_1 \ldots t_n]\!] & = c\, [\![t_1]\!] \ldots [\![t_n]\!] \\
(3) & [\![\lambda x.\, t]\!] & = \lambda x.\, [\![t]\!] \\
(4) & [\![e\langle f\rangle]\!] & = [\![e\langle t^f\rangle]\!] \\
(5) & [\![e\langle(\lambda x.\, t)\, t'\rangle]\!] & = [\![e\langle t\{x := t'\}\rangle]\!] \\
(6) & [\![e\langle \mathbf{let}\; x{=}t \;\mathbf{in}\; t'\rangle]\!] & = \mathbf{let}\; x{=}[\![t]\!] \;\mathbf{in}\; [\![e\langle t'\rangle]\!] \\
(7) & [\![e\langle \mathbf{case}\; x\, t_1 \ldots t_n \;\mathbf{of}\;\; q_1 \rightarrow s_1; \ldots; q_k \rightarrow s_k\rangle]\!] & \\
 & & = \mathbf{case}\; x\, [\![t_1]\!] \ldots [\![t_n]\!] \;\mathbf{of}\;\; q_1 \rightarrow [\![e\langle s_1\rangle]\!]; \ldots; q_k \rightarrow [\![e\langle s_k\rangle]\!] \\
(8) & [\![e\langle \mathbf{case}\; c\, t_1 \ldots t_n \;\mathbf{of}\;\; q_1 \rightarrow s_1; \ldots; q_k \rightarrow s_k\rangle]\!] & \\
 & & = [\![e\langle s_j\{x_1 := t_1, \ldots, x_n := t_n\}\rangle]\!]
\end{array}
$$

In (4) t^f is the right hand side of the definition among $d_1, \ldots, d_n$ with left hand side f. In (8) $q_j \equiv c\, x_1 \;\ldots x_n$. □

As is well-known, this algorithm hardly ever terminates. For instance, on the program $\mathbf{letrec}\, f = f\, \mathbf{in}\, f$ the same term f is encountered over and over again. To avoid this, the algorithm must incorporate *folding*, i.e., recall the terms it encounters and make repeating terms into recursive definitions.

Definition 8 (Folding) Let $[\![.]\!]$ take a parameter I (which will contain the set of previously encountered terms). $[\![.]\!]I$ is defined by Definition 7(1-3,5-8) (where I is passed unchanged to the recursive calls of $[\![.]\!]$) and in addition:

$$(0')\ [\![\textbf{letrec}\ d_1; \ldots ; d_n\ \textbf{in}\ t]\!] = [\![t]\!]\{\}$$

$$(4')\ [\![e\langle f\rangle]\!]I = \begin{cases} g\,x_1 \ldots x_n\ \textbf{if}\ g = \lambda x_1, \ldots, x_n.\, e\langle f\rangle \in I \\ g\,x_1 \ldots x_n\ \textbf{else, where} \\ \qquad\qquad I' = I \cup \{g = \lambda x_1, \ldots, x_n.\, e\langle f\rangle\} \\ \qquad\qquad g = \lambda x_1, \ldots, x_n.\, [\![e\langle t^f\rangle]\!]I' \end{cases}$$

where $\mathrm{FV}(e\langle f\rangle) = \{x_1, \ldots, x_n\}$ in some order. Now $[\![.]\!]$ applied to a program results in a term and a new set of definitions $g = \lambda x_1, \ldots, x_n. .[\![e\langle t^f\rangle]\!]I$ generated in the process, which are collected into a new program. □

It is interesting to note that on programs in the first-order fragment, the algorithm is equivalent to the usual first-order blazed deforestation algorithm.

Example 9 Recall the first program p in Example 1. With abbreviations:

$$I = \{da = \lambda us, vs, ws.\ a\ (a\ us\ vs)\ ws\}$$

$$I' = \{f = \lambda us, vs, ws.\ \textbf{case}\ (a\ us\ vs)\ \textbf{of}\ [\,] \rightarrow ws;\ (t\!:ts) \rightarrow t : a\ ts\ ws\}$$

$$I'' = \{a' = \lambda ys, zs.\ a\ ys\ zs\}$$

deforestation proceeds as follows:

$$
\begin{aligned}
\llbracket p \rrbracket &= \llbracket \lambda us, vs, ws.\ a\ (a\ us\ vs)\ ws \rrbracket \{\} && (3)\\
&= \lambda us, vs, ws.\ \llbracket a\ (a\ us\ vs)\ ws \rrbracket \{\} && (4')\\
&= \lambda us, vs, ws.\ da\ us\ vs\ ws\\
& && \textbf{where}\\
da &= \lambda us, vs, ws.\ \llbracket (\lambda xs, ys.\ \textbf{case}\ xs\ \textbf{of}\ [\,] \rightarrow ys;\ (t : ts) \rightarrow t : a\ ts\ ys)\ (a\ us\ vs)\ ws \rrbracket I && (5)\\
&= \lambda us, vs, ws.\ \llbracket \textbf{case}\ (a\ us\ vs)\ \textbf{of}\ [\,] \rightarrow ws;\ (t : ts) \rightarrow t : a\ ts\ ws \rrbracket I && (4')\\
&= \lambda us, vs, ws.\ f\ us\ vs\ ws\\
& && \textbf{where}\\
f &= \lambda us, vs, ws.\ \llbracket\ \textbf{case}\ \ ((\lambda xs, ys.\ \textbf{case}\ xs\ \textbf{of}\ [\,] \rightarrow ys;\ (s : ss) \rightarrow s : a\ ss\ ys)\ us\ vs)\ \textbf{of}\\
&\qquad\qquad [\,] \ \rightarrow ws\\
&\qquad\qquad (t : ts) \rightarrow t : a\ ts\ ws \rrbracket I \cup I' && (1, 2, 5, 7)\\
&= \lambda us, vs, ws.\ \textbf{case}\ \ us\ \textbf{of}\\
&\qquad\qquad [\,] \ \rightarrow \textbf{case}\ vs\ \textbf{of}\ [\,] \rightarrow ws;\ (t : ts) \rightarrow t : \llbracket a\ ts\ ws \rrbracket I \cup I'\\
&\qquad\qquad (t' : ts') \rightarrow t' : \llbracket a\ (a\ ts'\ vs)\ ws \rrbracket I \cup I' && (4')\\
&= \lambda us, vs, ws.\ \textbf{case}\ \ us\ \textbf{of}\\
&\qquad\qquad [\,] \ \rightarrow \textbf{case}\ vs\ \textbf{of}\ [\,] \rightarrow ws;\ (t : ts) \rightarrow t : a'\ ts\ ws\\
&\qquad\qquad (t' : ts') \rightarrow t' : da\ ts'\ vs\ ws\\
& && \textbf{where}\\
a' &= \lambda ts, ws.\ \llbracket \textbf{case}\ ts\ \textbf{of}\ [\,] \rightarrow ws;\ (t' : ts') \rightarrow t' : a\ ts'\ ws \rrbracket I \cup I' \cup I'' && (1, 2, 4', 7)\\
&= \lambda ts, ws.\ \textbf{case}\ ts\ \textbf{of}\ [\,] \rightarrow ws;\ (t' : ts') \rightarrow t' : a'\ ts'\ ws
\end{aligned}
$$

Hence the new program is

$$
\begin{array}{l}
\textbf{letrec} \\
\quad \begin{array}{lcl}
da & = & \lambda xs, ys, zs.\ f\ us\ vs\ ws \\
f & = & \lambda xs, ys, zs.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \qquad\qquad [\,] \quad \to \textbf{case}\ ys\ \textbf{of}\ [\,] \to zs;\ (t : ts) \to t : a'\ ts\ zs \\
 & & \qquad (t : ts) \to t : da\ ts\ ys\ zs \\
a' & = & \lambda ys, zs.\ \textbf{case}\ ys\ \textbf{of}\ [\,] \to zs;\ (t : ts) \to t : a'\ ts\ zs
\end{array} \\
\textbf{in}\ \lambda us, vs, ws.\ da\ us\ vs\ ws
\end{array}
$$

This is equivalent to the efficient program in Example 1. □

Definition 10 (Encountered terms) Given a program p, if $[\![p]\!] = \cdots [\![t]\!] \cdots$, then deforestation of p *encounters* $\lambda x_1, \ldots x_n.t$, where $\mathrm{FV}(t) = \{x_1, \ldots, x_n\}$. □

Example 11 Consider the transformation in Example 9. The first terms encountered by deforestation is $\lambda u, v, w.\, a\,(a\,u\,v)\,w$, and the next (different) term is $\lambda u, v, w.\,(\lambda x, y.\ \textbf{case}\ x\ \textbf{of}\ [\,] \to w;\ (h : t) \to h : a\ t\ y)\ (a\ u\ v)\ w$. □

Remark 12 Strictly speaking, in Definition 8 we should have replaced clause (5) of Definition 7 by a clause (5') analogous to how we replaced clause (4) of Definition 7 by (4'). The resulting algorithm can be proved to terminate whenever the one in Definition 7 encounters only finitely many different terms. However, in a number of situations, e.g. when the program is either linear or Hindley-Milner typable, this is not necessary. In what remains we simply assume that there is some way of folding such that if the algorithm in Definition 7 encounters only finitely many different terms then the algorithm extended with folding terminates. Our job, then, will be to make sure that the algorithm in Definition 7 encounters only finitely many different terms. □

Apart from termination—the topic of this paper—there are two other aspects of correctness for deforestation: *preservation of operational semantics* and *non-degradation of efficiency.*

As for preservation of operational semantics, the output of deforestation should be semantically equivalent to the input. That each step of the transformation rules for deforestation preserves call-by-need semantics is easily proved, but extending rigorously the proof to account for folding is more involved. A technique due to Sands [45,47] can be used to prove this for deforestation [44,46]. It is beyond the scope of the present paper to explain this technique.

As for non-degradation in efficiency, the output of deforestation should be at least as efficient as the input. There are several aspects of this problem. First, transformation can change a polynomial-time program into an exponential-time program by *computation duplication*. As mentioned, this may be avoided by considering only linear programs, or programs satisfying similar semantic conditions. Similar problems are addressed in partial evaluation [53,4]. Second, unfolding may increase the size of a program dramatically by *code duplication*. In principle the size of a program does not affect its running-time, but in practice this is not always true. Third, transformation steps can loose *laziness* and *full laziness*, as is described in detail by Marlow [37]. A proper development of these efficiency considerations is beyond the scope of this paper. This is not to suggest that these problems are not important; on the contrary, we believe that they are so important that they constitute separate problems.

4 Termination problems in deforestation

Even with folding, deforestation does not always terminate. In this section we present the three kinds of problems that may occur—we shall later see that these are the *only* problems that we have to consider. We show that deforestation of certain programs loops indefinitely, but with certain changes in the programs, deforestation terminates. These changes are called *generalizations*.

Example 13 (The Accumulating Parameter) Consider the program:

$$\begin{array}{l} \textbf{letrec} \\ \quad r \;\;= \lambda us.\, rr\; us\, [\,] \\ \quad rr = \lambda xs, ys.\, \textbf{case}\; xs\; \textbf{of}\;\; [\,] \to ys;\;\; (z : zs) \to rr\; zs\; (z : ys) \\ \textbf{in}\; r \end{array}$$

Here r returns its argument list reversed. Deforestation of this program loops indefinitely, because it encounters the progressively larger terms $\lambda zs_0.\, rr\, zs_0\, [\,]$, and $\lambda zs_1, z_1.\, rr\; zs_1\, [z_1]$, and $\lambda zs_2, z_2, z_1.\, rr\; zs_2\, [z_2, z_1]$, etc. Since parameter ys of rr is bound to progressively larger terms, Chin calls ys an *accumulating parameter*. We can solve the problem by forcing deforestation not to distinguish between the terms bound to ys. For this, we transform the program into:

$$\begin{array}{l} \textbf{letrec} \\ \quad r \;\;= \lambda us.\, rr\; us\, [\,] \\ \quad rr = \lambda xs, ys.\, \textbf{case}\; xs\; \textbf{of}\;\; [\,] \to ys;\;\; (z : zs) \to \textbf{let}\; v{=}z : ys\; \textbf{in}\; rr\; zs\; v \\ \textbf{in}\; r \end{array}$$

Deforestation applied to this program terminates. □

Example 14 (The Obstructing Function Call) Consider the program:

$$
\begin{array}{l}
\textbf{letrec} \\
\quad r \,=\, \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \quad\ \to [] \\
\qquad\qquad\qquad (z : zs) \to \textbf{case}\ (r\ zs)\ \textbf{of}\ [] \to [z];\ (y : ys) \to y : a\ ys\ [z] \\
\quad a \,=\, \lambda us, ws.\ \textbf{case}\ us\ \textbf{of}\ [] \to ws;\ (v : vs) \to v : a\ vs\ ws \\
\textbf{in}\ r
\end{array}
$$

The r function again reverses its argument, first reversing the tail and then appending the head. Deforestation encounters r, $\lambda zs_1, z_1.$ **case** $(r\,zs_1)$ **of** $\cdots$, $\lambda zs_2, z_2, z_1.$ **case** (**case** $(r\ zs_2)$ **of** $\cdots$) **of** $\cdots$, etc. Because the call to r prevents the surrounding case-expressions from being reduced, Chin calls it an *obstructing function call.* We can solve the problem by forcing deforestation not to distinguish between these terms. For this, we transform the program into:

$$
\begin{array}{l}
\textbf{letrec} \\
\quad r \,=\, \lambda xs.\ \textbf{let}\ l{=}r\ xs\ \textbf{in}\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad [] \quad\ \to [] \\
\qquad\qquad\qquad\qquad\qquad (z : zs) \to \textbf{case}\ l\ \textbf{of}\ [] \to [z];\ (y : ys) \to y : a\ ys\ [z] \\
\quad a \,=\, \lambda us, ws.\ \textbf{case}\ us\ \textbf{of}\ [] \to ws;\ (v : vs) \to v : a\ vs\ ws \\
\textbf{in}\ r
\end{array}
$$

Deforestation applied to this program terminates with the same program as output, which is satisfactory. $\Box$

Example 15 (The Accumulating Spine) Yet another possibility to prevent deforestation from termination is to create increasingly large spines of function applications. Consider the following program.

$$
\begin{array}{l}
\textbf{letrec} \\
\quad f \,=\, \lambda x.\ f\ x\ x \\
\textbf{in}\ f
\end{array}
$$

Note that such kind of function definitions is prohibited in first-order programs as well as in some typing disciplines. Deforestation applied to the program successively encounters terms f, $\lambda x.\ f\ x\ x$, $\lambda x.\ f\ x\ x\ x$, etc., and thus never terminates. The problem is resolved if we modify the program to:

$$\begin{array}{l}\textbf{letrec}\\ \;f \;=\; \lambda x.\, (\textbf{let}\; y{=}f\; \textbf{in}\; y\; x)\; x\\ \textbf{in}\; f\end{array}$$

Then deforestation terminates. □

The operation that replaces $t\{x := t'\}$ by **let** $x{=}t'$ **in** t is called *generalization of* t' *at* $t\{x := t'\}$; it occurs in variations in several transformation techniques. In Example 13, we generalized rr's second argument $z : ys$ at the application $rr\; zs\, (z : ys)$ in the body of the definition for rr. In example 14, we generalized the call $r\, xs$ at the case-expression in the body of the definition of r. In example 15, we generalized f at the application $f\; x$ in the body of the definition of f. Generalizing should be thought of as blazing. Instead of putting marks on our programs we introduce let-expressions.

The syntactic techniques of Wadler, Chin, and Hamilton solve the two types of problems in first-order deforestation—accumulating parameters and obstructing function calls—by requiring function arguments and case-selectors to be variables; this is what the treeless syntax enforces. In the higher-order case, the last type of problem—accumulating spines—is solved in the techniques by Hamilton and Marlow by assuming programs to be Hindley-Milner typable.

5 Constraint systems

In this section we review the necessary constraint terminology for the constraint-based approximation of higher-order deforestation.

Let D be a complete lattice and V a set of variables. We consider constraints

$$X_0 \sqsupseteq f X_1 \ldots X_n$$

where $X_i \in V$ and f denotes a monotonous function $[\![f]\!] : D^n \;\to\; D$. A set S of such constraints has a least model $\mu S : V \to D$ such that

$$(\mu S\ X) \sqsupseteq [\![f]\!](\mu S\ X_1) \ldots (\mu S\ X_n)$$

for every constraint $X \sqsupseteq f X_1 \ldots X_n \in S$.

We use two instances of this scheme:*simple constraints* and *integer constraints*. In a set of simple constraints, a finite set A of objects is given. D is the powerset of A ordered by set inclusion. In our application we need no occurrences of variables or operators in right-hand sides of simple constraints (but we do in integer constraints—see below), so they are of the simple form $X \supseteq a$ for some $a \in A$. One important special case of simple constraints is given by a one-element set A. Then 2^A is isomorphic to the 2–point domain $\mathbf{2} = \{0 \sqsubset 1\}$. These constraints are called *boolean*.

In integer constraints D is the non-negative numbers $\mathcal{N}$ with their natural ordering and extended by ∞. Right hand sides are built from variables and constants by means of certain operations, in our case "+" and "$\sqcup$" (maximum).

Example 16 Consider the integer constraints:

$$X \geq 1 \qquad Z \geq X + Y \quad Y \geq 7 \qquad Y \geq Z \sqcup X$$

In the least model, $\mu S\ X = 1, \mu S\ Y = \mu S\ Z = \infty$. $\Box$

$\mathcal{N}$ does not satisfy the ascending chain condition, so naive fix-point iteration may not suffice to compute least models. Seidl [49,51] presents algorithms that do compute such least models, [1] in our case in linear time.

The following result will be useful later on.

Proposition 17 *Let S_1 and S_2 be sets of constraints with variables from V over complete lattice D. If $S_1 \subseteq S_2$ then $\mu S_1\ X \sqsubseteq \mu S_2\ X$, for all $X \in V$.* $\Box$

6 Approximating deforestation by constraints

In this section we show how to compute, for a given program, a set of integer constraints whose least model indicates which subterms cause termination problems. The next definition is followed by extensive explanations.

Definition 18 (Approximation of deforestation) Given some program

$$p \equiv \mathbf{letrec}\ d_1; \ldots ; d_n\ \mathbf{in}\ t_{init}$$

Let T be the set of subterms of t_{init} and of the right hand sides of definitions in p, and $T_\bullet = T \cup \{\bullet\}$, where $\bullet$ is some symbol. We use variables $[p]$ and $[t]$, $r[t]$, $d[t]$, $s[t]$ and $a[t]$ for all $t \in T$. The set of constraints $\mathcal{C}(p)$ for p is the smallest set containing the *initial* constraints, and closed under the *transitivity rules*, *top-level rules*, and *unfolding rules*. $\mathcal{I}(p)$ is the set of integer constraints among $\mathcal{C}(p)$.

[1] Seidl [49,51] considers *equations*. This makes no difference w.r.t. minimal models.

Initial constraints:

$[p] \supseteq t_{init}; [t] \supseteq t; a[t] \geq N[t]$

where

$$
\begin{array}{ll}
N[x] & = a[x] \\
N[f] & = 0 \\
N[c\, t_1 \ldots t_n] & = 1 + (N[t_1] \sqcup ... \sqcup N[t_n]) \\
N[t_1\, t_2] & = 1 + (N[t_1] \sqcup N[t_2]) \\
N[\textbf{let}\ x{=}t\ \textbf{in}\ t'] & = 1 + (N[t] \sqcup N[t']\{a[x] := 0\}) \\
N[\lambda x.\, t] & = 1 + N[t]\{a[x] := 0\} \\
N[\textbf{case}\ t_0\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m] & = 1 + (N[t_0] \sqcup N_{q_1}[t_1] \sqcup \ldots \sqcup N_{q_m}[t_m]) \\
N_{c\, x_1 \ldots x_n}[t] & = N[t]\{a[x_1] := 0, \ldots, a[x_n] := 0\}
\end{array}
$$

Transitivity rules:

if $[p] \supseteq t', [t'] \supseteq t''$ then $[p] \supseteq t''$

if $[t] \supseteq t', [t'] \supseteq t''$ then $[t] \supseteq t''$

if $r[t] \sqsupseteq 1, [t] \supseteq t'$ then $r[t'] \sqsupseteq 1; d[t'] \geq d[t]; s[t'] \geq s[t]$

Top-level rules:

if $[p] \supseteq c\, t_1 \ldots t_n$ then $[p] \supseteq t_1, \ldots, t_n$

if $[p] \supseteq \lambda x.\, t$ then $[p] \supseteq t, [x] \supseteq \bullet$

if $[p] \supseteq t$ then $r[t] \sqsupseteq 1$

Unfolding rules:

if $r[t] \sqsupseteq 1$ then case t of

f : $[t] \supseteq t^f$;

$t_1\, t_2$:

$r[t_1] \sqsupseteq 1; s[t_1] \geq 1 + s[t]; d[t_1] \geq d[t]$;

if $[t_1] \supseteq \lambda x.t'$ then $[t] \supseteq t'; [x] \supseteq t_2; a[x] \geq a[t_2]$;

if $[t_1] \supseteq \bullet$ then $[t] \supseteq \bullet; [p] \supseteq t_2$;

$\textbf{let}\ x{=}t_1\ \textbf{in}\ t_2$:

$[t] \supseteq t_2; [x] \supseteq \bullet; [p] \supseteq t_1$;

$\textbf{case}\ t_0\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m$:

$r[t_0] \sqsupseteq 1; s[t_0] \geq s[t]; d[t_0] \geq 1 + d[t]$;

if $[t_0] \supseteq \bullet$ then

$[t] \supseteq t_1, \ldots, t_m; [y] \supseteq \bullet$ (for all y in $q_1, \ldots, q_m$);

if $[t_0] \supseteq c\, s_1 \ldots s_n$ and $q_j \equiv c\, x_1 \ldots x_n$ then

$[t] \supseteq t_j; [x_1] \supseteq s_1; \ldots; [x_n] \supseteq s_n; a[x_1] \geq a[s_1]; \ldots; a[x_n] \geq a[s_n]$; □

The constraints can be divided into five disjoint groups, each making use of the types of variables depicted in the following table.

Variables	Lattice	Constraint Type	Constraint form
$[p], [t]$	$2^{T_\bullet}$	simple	$[p] \supseteq t, [p] \supseteq \bullet, [t] \supseteq t', [t] \supseteq \bullet$
$r[t]$	$\mathbf{2}$	boolean	$r[t] \sqsupseteq 1$
$d[t]$	$\mathcal{N}$	integer	$d[t] \geq d[t'], d[t] \geq 1 + d[t']$
$s[t]$	$\mathcal{N}$	integer	$s[t] \geq s[t'], s[t] \geq 1 + s[t']$
$a[t]$	$\mathcal{N}$	integer	$a[t] \geq a[t'], a[t] \geq N[t]$

The meaning of the variables are as follows.

(i) The *simple* variable $[p]$ represents a superset of the terms encountered when deforesting p. The simple variable $[t]$ represents a superset of the terms encountered when deforesting t. The symbol $\bullet$ denotes a term of form $x\, t_1 \ldots t_n$; such terms block the unfolding of applications and case-expressions when they are operator and selector, respectively.
(ii) The *boolean* variable $r[t]$ shows whether transformation of t is forced by a surrounding context.
(iii) The *integer* expression $s[t] + d[t]$ is an upper bound for the depth of contexts in which t occurs during transformation; $s[t]$ and $d[t]$ count the nesting inside operators of applications and selectors of case-expressions, respectively. The variable $a[x]$ is an upper bound for the depth of terms bound to x during deforestation; $a[t]$ is an upper bound for the depth of t during transformation, taking binding of variables in t into account.

The effect of the constraints are as follows.

(i) The *initial constraints* $[t] \supseteq t$ model reflexivity of iterated unfolding, $[p] \supseteq t_{init}$ shows that the main term t_{init} is due for transformation, and $a[t] \geq N[t]$ gives lower bounds for $a[t]$. Since t may contain free variables from some set $\mathcal{V}$, $N[t]$ is a polynomial over $a[x]$, $x \in \mathcal{V}$.
(ii) The *transitivity rules* model iterated unfolding.
(iii) The *top-level* rules model rules (2-3) of deforestation which push transformation under constructors and abstractions (rule (1) is modeled by the unfolding rules). When going under an abstraction, the abstracted variable obtains the status of a free variable and so receives value $\bullet$.
(iv) The *unfolding rules* model (4-8) of deforestation which unfold redexes.
First, the rules model individual steps of the deforestation process, i.e., expansion of function names and reduction of β-redexes and case-expressions with new bindings for substituted variables.

Second, information about which subterm is unfolded next by deforestation is propagated, i.e., the $r[.]$-information is propagated to the function part of applications and to the selector of case-expressions.

Third, transformation of certain subterms of let-expressions and case-expressions must be raised to the top-level. Also, the information involving $\bullet$ must be propagated, reflecting rule (1).

Finally, information about the depth of contexts and arguments is recorded. Constraints with $a[.]$-variables are generated when variables become bound by reductions of β-redexes and case-expressions, and constraints with $s[.]$- and $d[.]$-variables are generated when passing control to the function part of an application or the selector of a case-expression.

Example 19 The program in Example 15 has constraints:

$$[p] \supseteq f, \lambda x.\ f\ x\ x, f\ x\ x$$

$$\begin{array}{llll}
r[x] & \sqsupseteq & [x] & \supseteq x \\
r[f] & \sqsupseteq 1 & [f] & \supseteq f, \lambda x.\ f\ x\ x \\
r[f\ x] & \sqsupseteq 1 & [f\ x] & \supseteq f\ x, f\ x\ x \\
r[f\ x\ x] & \sqsupseteq 1 & [f\ x\ x] & \supseteq f\ x\ x \\
r[\lambda x.\ f\ x\ x] & \sqsupseteq 1 & [\lambda x.\ f\ x\ x] & \supseteq \lambda x.\ f\ x\ x
\end{array}$$

$$\begin{array}{llll}
a[x] & \geq a[x] & d[x] & \geq \\
a[f] & \geq 0 & d[f] & \geq d[f], d[f\ x] \\
a[f\ x] & \geq 1 + a[x] & d[f\ x] & \geq d[f\ x], d[f\ x\ x] \\
a[f\ x\ x] & \geq 2 + a[x] & d[f\ x\ x] & \geq d[f\ x\ x], d[f\ x] \\
a[\lambda x.\ f\ x\ x] & \geq 3 & d[\lambda x.\ f\ x\ x] & \geq d[\lambda x.\ f\ x\ x], d[f]
\end{array}$$

$$\begin{array}{ll}
s[x] & \geq \\
s[f] & \geq s[f], 1 + s[f\ x] \\
s[f\ x] & \geq s[f\ x], 1 + s[f\ x\ x] \\
s[f\ x\ x] & \geq s[f\ x\ x], s[f\ x] \\
s[\lambda x.\ f\ x\ x] & \geq s[\lambda x.\ f\ x\ x], s[f]
\end{array}$$

The integer constraints I include:

$$s[f] \geq 1 + s[f\ x] \qquad s[f\ x] \geq 1 + s[f\ x\ x] \qquad s[f\ x\ x] \geq s[f\ x]$$

In particular $\mu I\ s[f] = \infty$, reflecting the fact that transformation encounters terms with f embedded in unboundedly deep applications. □

Example 20 The program in Example 13 has integer constraints I including:

$$a[ys] \geq a[z : ys] \geq 1 + a[ys]$$

In particular $\mu I\ a[ys] = \infty$, reflecting the fact that transformation encounters terms with unboundedly large arguments containing ys. $\square$

Example 21 The program in Example 14 has integer constraints I including:

$$\begin{aligned} d[r\ zs] &\geq 1 + d[\mathbf{case}\ (r\ zs) \ldots] \\ d[\mathbf{case}\ (r\ zs) \ldots] &\geq d[r\ zs] \end{aligned}$$

which imply $\mu I\ d[r\ zs] = \infty$, reflecting the fact that transformation encounters terms with calls $r\ zs$ embedded in unboundedly deep case-expressions. $\square$

Computing set $\mathcal{C}(p)$ can be viewed as a *control-flow analysis* in the sense of [57,58,42,43] resp. *closure analysis* in the sense of [54,5] extended to a language with case-expressions and adapted to an outermost unfolding strategy. It is closely related to normalization of set constraints as considered by Heintze [27] for a call-by-value language. While performing control-flow analysis of program p, we keep track of the depth of applications and case-expressions in which unfolding occurs and the depth of arguments bound to formal parameters by means of integer constraints.

7 Complexity and correctness of the analysis

This section estimates the complexity of computing $\mathcal{C}(p)$ together with the least model of $\mathcal{I}(p)$, the set of integer constraints contained in $\mathcal{C}(p)$. Moreover, a proof is presented showing that the constraints in $\mathcal{I}(p)$ contain enough information to approximate whether deforestation stops. Both claims are made precise in the following theorem:

Theorem 22 *Let* $p \equiv \mathbf{letrec}\ d_1; \ldots; d_n\ \mathbf{in}\ t_{init}$ *and* $I = \mathcal{I}(p)$.

(i) *If deforestation of p encounters infinitely many different terms, then:*
  (1) $\mu I\ a[x] = \infty$ *for some variable x; or*
  (2) $\mu I\ d[t] = \infty$ *for some subterm t; or*
  (3) $\mu I\ s[t] = \infty$ *for some subterm t.*
(ii) *I and μI can be computed in time* $\mathcal{O}(|p|^3)$.

Here (1-2) correspond to Sørensen's [59,50] criteria in first-order deforestation for accumulating parameters and obstructing function calls, respectively. In the higher-order case, (3) captures accumulating spines. In Section 9 we show that monomorphically typed programs never give rise to accumulating spines.

Proof. (Theorem 22(ii).) To compute $\mathcal{C}(p)$ we employ a version of Heintze's cubic-time algorithm for computing a normalized system of set constraints [27], where right-hand sides in constraints $X \supseteq t$ are no longer viewed as expressions built up from set operations, but simply as singleton sets $\{t\}$ of atoms. The additional constraints for variables $r[.]$ are used to model call-by-name evaluation (instead of call-by-value evaluation in [27]). The computed subset $\mathcal{I}(p)$ of integer constraints contains at most $|p|^2$ constraints. Then we use Seidl's algorithm [49,51] to compute the least model of $\mathcal{I}(p)$ in time linear in the size of $\mathcal{I}(p)$ (cf. Prop. 39). $\square$

Cubic time for normalization of set constraints is a merely theoretical worst-case estimation. For his applications, Heintze found his algorithm to behave much better in practice [27].

The rest of this section is devoted to the proof of part (i) of Theorem 22. Our methods extend those used in [50] to the higher-order case. First we set up a rewrite relation $\Rightarrow$ which gives a more convenient way to deal with the notion of transformation encountering terms (Def. 23). Then we rephrase this rewrite relation by means of stacks and environments to make substitutions explicit (Def. 26). The latter formulation is is well-suited for abstraction through sets of constraints (Def. 30). It is then proved that the set of constraints generated by our algorithm safely approximates the abstraction of every stack derivable from p (Prop. 34). Furthermore, we have to verify that the generated set of integer constraints indeed allows to extract information about termination of deforestation (Props. 33, 35, 37). From this, we deduce the correctness of our analysis.

Definition 23 Define $\Rightarrow$ on programs and closed terms by:

$$
\begin{array}{lll}
(0) & \textbf{letrec}\, d_1; \ldots; d_n \,\textbf{in}\, t & \Rightarrow t \\
(1a) & e\langle \bullet\, t \rangle & \Rightarrow e\langle \bullet \rangle \\
(1b) & e\langle \bullet\, t \rangle & \Rightarrow t \\
(2) & c\, t_1 \ldots t_n & \Rightarrow t_i \quad (\text{all } i) \\
(3) & \lambda x.\, t & \Rightarrow t^{\bullet} \\
(4) & e\langle f \rangle & \Rightarrow e\langle t^f \rangle \quad (f = t^f) \\
(5) & e\langle (\lambda x.\, t)\, t' \rangle & \Rightarrow e\langle t\{x := t'\} \rangle \\
(6a) & e\langle \textbf{let}\, x{=}t' \,\textbf{in}\, t \rangle & \Rightarrow e\langle t^{\bullet} \rangle \\
(6b) & e\langle \textbf{let}\, x{=}t' \,\textbf{in}\, t \rangle & \Rightarrow t' \\
(7) & e\langle \textbf{case}\ \bullet\ \textbf{of}\ \ q_1 \to s_1; \ldots; q_m \to s_m \rangle & \Rightarrow e\langle s_i^{\bullet} \rangle \quad (\text{for all } i) \\
(8) & e\langle \textbf{case}\ (c\, t_1 \ldots t_n)\ \textbf{of}\ \ q_1 \to s_1; \ldots; q_m \to s_m \rangle & \Rightarrow e\langle s_j\{x_1{:=}t_1, \ldots, x_n{:=}t_n\} \rangle
\end{array}
$$

Here $t^{\bullet}$ denotes the result of replacing all free variables of t by $\bullet$. In (4) definitions $f = t^f$ must be located among $d_1, \ldots, d_n$ of the program, and in (8), $q_j \equiv c\, x_1\ \ldots x_n$. $\square$

These rules correspond to rules (0-8) of deforestation. Each of the rules except (1a,1b,6a,6b,7) correspond to the similar rule in deforestation; (6a,6b) together correspond to (6). Rules (1a,1b,7) together correspond to (1,7) in deforestation; terms of form $\bullet\, t_1 \ldots t_n$ are successively rewritten to $\bullet$ by (1a) while the t_i are raised to the top-level by (1b). This not only corresponds to rule (1) of deforestation but has the additional advantage that in our new rule (7), we only need to consider selector $\bullet$ and not arbitrary selectors $x\, t_1 \ldots t_m$.

The following shows that the problem of ensuring that $[\![.]\!]$ of Definition 7 encounters only finitely many different terms (see Remark 12) is equivalent to the problem of ensuring that $p \stackrel{*}{\Rightarrow} s$ for only finitely many different s.

Proposition 24 *For any p, $p \stackrel{*}{\Rightarrow} s$ iff $[\![p]\!] = \cdots [\![t]\!] \cdots$ for some t with $t^{\bullet} \equiv s$.*

Proof. "if' is by induction on the definition of $[\![.]\!]$, and "only if" is by induction on the number of steps in $\stackrel{*}{\Rightarrow}$. $\square$

Instead of reducing terms as suggested by the deforestation algorithm, we prefer (just for the correctness proof of the analysis) to manipulate substitutions and call-by-name contexts explicitly by means of environments and stacks, respectively.

Definition 25 Let E and σ range over *environments* and *stacks*, respectively, as defined by the following grammar (where $n \geq 0$):

$$\begin{aligned}
E &::= \{x_1 := (t_1, E_1), \ldots, x_n := (t_1, E_n)\} \\
\sigma &::= (t, E)\ \tau_1 \ldots \tau_n \\
\tau &::= (t\, t', E) \mid (\textbf{case}\ t_0\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m, E)
\end{aligned}$$

The length of a stack is defined by $\|(t, E)\tau_1 \ldots \tau_n\| = n$. Also, $\emptyset$ is the empty environment, and $E_1 + E_2$ is the concatenation of environments E_1, E_2. $\square$

On stacks we introduce "$\to$" which simulates the relation "$\Rightarrow$" on terms.

Definition 26 Define $\to$ on programs, pairs (t, E) and stacks σ by:

$$\begin{array}{lll}
(0a) & \textbf{letrec}\, d_1 \ldots d_n\, \textbf{in}\, t & \to (t, \emptyset) \\
(0b) & (x, E)\ \sigma & \to (E(x))\ \sigma \\
(0c) & (t\, t', E)\ \sigma & \to (t, E)\ (t\, t', E)\ \sigma \\
(0d) & (\textbf{case}\ t_0\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m, E)\ \sigma & \\
 & & \to (t_0, E)\ (\textbf{case}\ t_0\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m, E)\ \sigma \\
(1a) & (\bullet, \emptyset)\ (t\, t', E)\ \sigma & \to (\bullet, \emptyset)\ \sigma \\
(1b) & (\bullet, \emptyset)\ (t\, t', E)\ \sigma & \to (t', E) \\
(2) & (c\, t_1 \ldots t_n, E) & \to (t_i, E) \quad (\text{all } i) \\
(3) & (\lambda x.\, t, E) & \to (t, \{x := (\bullet, \emptyset)\} + E) \\
(4) & (f, E)\ \sigma & \to (t^f, \emptyset)\ \sigma \\
(5) & (\lambda x.\, t, E)\ (t_1\, t_2, E_2)\ \sigma & \to (t, E + \{x := (t_2, E_2)\})\ \sigma \\
(6a) & (\textbf{let}\ x{=}t\ \textbf{in}\ t', E)\ \sigma & \to (t', E + \{x := (\bullet, \emptyset)\})\ \sigma \\
(6b) & (\textbf{let}\ x{=}t\ \textbf{in}\ t', E)\ \sigma & \to (t, E) \\
(7) & (\bullet, \emptyset)\ (\textbf{case}\ t_0\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m, E_1)\ \sigma & \\
 & & \to (t_i, \{x_1 {:=} (\bullet, \emptyset), \ldots, x_n {:=} (\bullet, \emptyset)\} + E_1)\ \sigma \\
 & & \quad (\text{all } i,\ q_i \equiv c\, x_1 \ldots x_n) \\
(8) & (c\, s_1 \ldots s_n, E)\ (\textbf{case}\ t_0\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m, E_1)\ \sigma & \\
 & & \to (t_j, E_1 + \{x_1 := (s_1, E), \ldots, x_n := (s_n, E)\})\ \sigma \\
 & & \quad (q_j \equiv c\, x_1 \ldots x_n)
\end{array}$$

$\square$

Rules (0a,1a-8) simulate the corresponding rules of Definition 23. The rule (0b) is necessary to obtain the binding of a variable from the environment, whereas (0c,0d) are necessary to descend into a context to reach the redex which is then transformed at the top of the stack. The intuition behind a stack like $(t, E)(t_1\, t_2, E')$ is that t is the result of reducing t_1 a number of steps. By (5), once t is has been reduced to an abstraction $\lambda x.\, t'$, (t, E) can be popped and transformation proceed with t', with $\{x := (t_2, E')\}$ in the environment.

The following function recovers from a pair consisting of a term and an environment (or a stack of such pairs) the term that the pair (the stack) denotes.

Definition 27

$$
\begin{array}{ll}
u[(\bullet, E)] & = \bullet \\
u[(f, E)] & = f \\
u[(c\, t_1 \ldots t_n, E)] & = c\, u[(t_1, E)] \ldots u[(t_n, E)] \\
u[(\lambda x.\, t, E)] & = \lambda x.\, u[(t, E)] \\
u[(t\, t', E)] & = u[(t, E)]\, u[(t', E)] \\
u[(\mathbf{let}\ x{=}t\ \mathbf{in}\ t', E)] & = \mathbf{let}\ x{=}u[(t, E)]\ \mathbf{in}\ u[(t', E)] \\
u[(\mathbf{case}\ t_0\ \mathbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m, E)] & \\
 & = \mathbf{case}\ u[(t_0, E)]\ \mathbf{of}\ \ q_1 \to u[(t_1, E)]; \ldots; q_m \to u[(t_m, E)] \\
u[(x, E)] & = u[E\ (x)] \ \text{ if } x \in \mathit{dom}(E) \text{ and } x \text{ otherwise} \\
u[(t, E)(t_1\, t_2, E')\ \sigma'] & = u[(v\, t_2, E')\ \sigma'] \ \text{ if } v = u[(t, E)] \\
u[(t, E)(\mathbf{case}\ t_0\ \mathbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m, E')\ \sigma'] & \\
 & = u[(\mathbf{case}\ v\ \mathbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m, E')\ \sigma'] \ \text{ if } v = u[(t, E)]
\end{array}
$$
□

Example 28 Consider term s, environment E and stack σ given by

$$
\begin{aligned}
s &= \mathbf{case}\ z\ \mathbf{of}\ \ [] \to [];\ h : t \to h : I\ t \\
E &= \{z := (I\ t, \{t := (\bullet, \emptyset)\})\} \\
\sigma &= (h : t, \{h := (\bullet, \emptyset), t := (\bullet, \emptyset)\})(s, E)
\end{aligned}
$$

Then

$$u[(h : t, \{h := (\bullet, \emptyset), t := (\bullet, \emptyset)\})] = \bullet : \bullet$$

Therefore,

$$
\begin{aligned}
u[\sigma] &= u[(h : t, \{h := (\bullet, \emptyset), t := (\bullet, \emptyset)\})\ (s, E)] \\
&= u[(\mathbf{case}\ \bullet : \bullet\ \mathbf{of}\ \ [] \to [];\ h : t \to h : I\ t, E)] \\
&= \mathbf{case}\ \bullet : \bullet\ \mathbf{of}\ \ [] \to [];\ h : t \to h : I\ t
\end{aligned}
$$
□

Proposition 29 *Assume $t = u[\sigma]$. Then*

(i) $t \Rightarrow t'$ implies $\sigma \xrightarrow{*} \sigma'$ for some σ' with $u[\sigma'] = t'$;
(ii) $\sigma \rightarrow \sigma'$ implies $t = u[\sigma']$ or $t \Rightarrow u[\sigma']$.

Proof. (i) is by induction on the definition of $t \Rightarrow t'$, (ii) is by induction on the definition of $\sigma \rightarrow \sigma'$. $\square$

We aim to show that the set of constraints $\mathcal{C}(p)$ for a program p gives certain information about the structure of the stacks σ such that $p \xrightarrow{*} \sigma$, expressed in Propositions 33 and 34 below. To this end we introduce a mapping α which *abstracts* stacks by a set of simple, boolean and integer constraints.

Specifically, a variable binding $x := (t, E)$ is recorded by constraints $[x] \supseteq t$ and $a[x] \geq N[t]$. The fact that $p \xrightarrow{*} (t, E)$ is recorded by $[p] \supseteq t$. Also, assume $p \xrightarrow{*} (t_0, E_0)\ (t_1\ t_1', E_1)\ \sigma$. Then transformation of t_1 is forced by the context, and this led to the stack $(t_1, E_1)\ (t_1\,t_1', E_1)\ \sigma$, which after a number of steps has become $(t_0, E_0)\ (t_1\ t_1', E_1)\ \sigma$. This is all expressed by the constraints $\{[t_1] \supseteq t_0, r[t_0] \sqsupseteq 1\}$. The integer constraints $s[t_0] \geq s[t_1] \geq 1 + s[t_1\ t_1']$ record the increase in the number of applications on the stack whereas the integer constraints $d[t_0] \geq d[t_1] \geq d[t_1\ t_1']$ record the fact that no increase in the number of case-expressions occurred. Case-expressions on the stack are treated analogously.

Definition 30 On environments and stacks define α as follows.

$$
\begin{array}{ll}
\alpha\emptyset & = \emptyset \\
\alpha(\{x{:=}(\bullet, \emptyset)\} + E) & = \{[x] \supseteq \bullet\} \cup \alpha E \\
\alpha(\{x{:=}(t, E_1)\} + E_2) & = \{[x] \supseteq t, a[x] \geq a[t], a[t] \geq N[t]\} \cup \alpha E_1 \cup \alpha E_2 \\
\alpha\epsilon & = \emptyset \\
\alpha(t, E) & = \{[p] \supseteq t, r[t] \sqsupseteq 1\} \cup \alpha E \\
\alpha((t, E)\ (t', E')\ \sigma) & = \{[t_0] \supseteq t, r[t] \sqsupseteq 1\} \\
& \quad \cup\ \{s[t] \geq s[t_0], s[t_0] \geq s[t'], d[t] \geq d[t_0], d[t_0] \geq 1 + d[t]\} \\
& \quad \cup\ \alpha E \cup \alpha((t', E')\ \sigma) \quad \text{if } t' \equiv \textbf{case}\ t_0\ \textbf{of}\ q_1 \rightarrow t_1; \ldots; q_m \rightarrow t_m \\
\alpha((t, E)\ (t', E')\ \sigma) & = \{[t_0] \supseteq t, r[t] \sqsupseteq 1\} \\
& \quad \cup\ \{s[t] \geq s[t_0], s[t_0] \geq 1 + s[t'], d[t] \geq d[t_0], d[t_0] \geq d[t]\} \\
& \quad \cup\ \alpha E \cup \alpha((t', E')\ \sigma) \quad \text{if } t' \equiv t_0\ t_1
\end{array}
$$
$\square$

Example 31 Consider again stack σ from Example 28. The set of simple constraints in $\alpha\sigma$ consists of:

$$[p] \supseteq s, [z] \supseteq I\, t, [h] \supseteq \bullet, [t] \supseteq \bullet, [z] \supseteq h : t \qquad \square$$

Definition 32 Define the *depth* $\|t\|$ of a term t as follows.

$$\begin{aligned}
\| \bullet \| = \|x\| = \|f\| &= 0 \\
\|c\, t_1 \ldots t_n\| &= 1 + (\|t_1\| \sqcup \ldots \sqcup \|t_n\|) \\
\|\mathbf{case}\ t\ \mathbf{of}\ \ q_1 \to t_1; \ldots; q_k \to t_k\| &= 1 + (\|t\| \sqcup \|t_1\| \sqcup \ldots \sqcup \|t_k\|) \\
\|\lambda x.\, t\| &= 1 + \|t\| \\
\|t\, t'\| = \|\mathbf{let}\ v{=}t\ \mathbf{in}\ t'\| &= 1 + (\|t\| \sqcup \|t'\|) \qquad \square
\end{aligned}$$

By induction on the structure of environments E and stacks σ we verify:

Proposition 33 *Let I, J be the integer constraints in αE, $\alpha\sigma$, respectively.*

(i) $\mu I\ a[x] \geq \|u[E(x)]\|$ *for every* $x \in \mathrm{dom}(E)$.
(ii) If $\sigma = (t, E)\ \sigma'$, *then* $(\mu J\ d[t]) + (\mu J\ s[t]) \geq \|\sigma\|$. $\square$

Furthermore, by induction on the length of $p \xrightarrow{*} \sigma$:

Proposition 34 *Given program p. If $p \xrightarrow{*} \sigma$ then $\alpha\sigma \subseteq \mathcal{C}(p)$.* $\square$

Finally, by induction on the definition of u, we verify the following proposition which relates the depth of terms, stacks, and environments.

Proposition 35 *Let $u[(t_0, E_0) \ldots (t_k, E_k)] = t$. Suppose that $\|t_j\| \leq r$ and $\|u[E_j(x)]\| \leq a$ for all j and $x \in \mathrm{dom}(E_j)$. Then $\|t\| \leq r + a + k$.* $\square$

Example 36 Consider term s, environment E and stack σ as in Example 28. The depth of σ is 1, the maximal depth of the t_i is 3, and the maximal value $\|u[E_j(x)]\|$ is given by $\|u[E(z)]\| = \|I \bullet\| = 1$. Thus, the upper bound computed by Proposition 35 equals $3 + 1 + 1 = 5$. Now, $\|u[\sigma]\| = 3$ which is indeed less than or equal to 5. $\square$

Proposition 37 *Given $p \equiv$ **letrec** $f_1 = t_1; \ldots; f_n = t_n$ **in** t_{init} and $I = \mathcal{I}(p)$. Let $r = \max\{\|t_{init}\|, \|t_1\|, \ldots, \|t_n\|\}$, and let a, d, s denote the maximal values of $\mu I\, a[x]$, $\mu I\, d[t']$, and $\mu I\, s[t']$, respectively. If $p \xrightarrow{*} \sigma$ then $\|u[\sigma]\| \leq r{+}a{+}d{+}s$.*

Proof. By Proposition 34, the set of integer constraints of $\alpha\sigma$ are contained in I. Hence by Propositions 17 and 33, $\|\sigma\| \leq d + s$ and $\|u[E(x)]\| \leq a$ for variables x and environments E occurring in σ. For every (t, E) somewhere in σ, t is a subterm of p, so by Proposition 35, $\|u[\sigma]\| \leq r + a + d + s$. $\square$

We are now in a position to prove the first part of Theorem 22.

Proof. (Theorem 22, first part). Assume that $[\![p]\!] = \cdots [\![t]\!] \cdots$. By Proposition 24, $p \stackrel{*}{\Rightarrow} t^\bullet$. Then by Proposition 29, $p \stackrel{*}{\to} \sigma$ for some stack σ with $u[\sigma] = t^\bullet$. By Proposition 37, $\|t\| = \|t^\bullet\| = \|u[\sigma]\| \leq r + a + d + s$, where a, d, s denote the maximal values of $\mu I\ a[x], \mu I\ d[t']$, and $\mu I\ s[t']$, respectively, and r denotes the maximal depth of the main term and right hand sides of definitions in p.

Now, if $[\![p]\!]$ encounters infinitely many different terms, $[\![p]\!]$ must encounter arbitrarily deep terms (there are only finitely many different closed terms of a given depth). The above then shows that one of a, d, s must be ∞. $\square$

8 Generalizing dangerous subterms

Theorem 22 shows how to guarantee that deforestation terminates on some program: check that conditions (1)-(3) are all false. It remains to compute appropriate generalizations in case one of the conditions are true, i.e., when deforestation may fail to terminate. In order to do so, we examine how infinity arises in the integer constraints generated by our analysis.

Given p and $I = \mathcal{I}(p)$. Any set of inequalities of form $Y \geq P$ in I, where P is a polynomial built from variables, constants, "$+$", "$\sqcup$", can be transformed into a set of constraints of the forms $Y \geq c + X$ and $Y \geq c$, where $c \geq 0$ is an integer, with the same minimal model. Hence, we may assume that all the constraints in I are of these forms.

Next we give a characterization of the variables X with $\mu I\ X = \infty$. Moreover, we make explicit how the set of these variables can be determined efficiently. The characterization is given in terms of *strong components* of the *dependence graph* of I.[2]

Definition 38 (Dependence graph) Given program p and $I = \mathcal{I}(p)$. The *dependence graph* G_I is the directed graph whose nodes are the variables of I, and whose edges are all (X, Y) with $Y \geq c + X \in I$. $\square$

[2] As observed in [50], one may also determine from μI the variables $a[t]$, $d[t]$, and $s[t]$ whose values exceed some threshold. This may be useful for preventing code explosion during deforestation, see [37].

A strong component Q of a directed graph G is a maximal subset of nodes of G such that there is a path in G from v_1 to v_2 for any nodes $v_1, v_2 \in Q$.

Proposition 39 *Let $I = \mathcal{I}(p)$ and $J_\tau = \{t \mid \mu I\ \tau[t] = \infty\}$, $\tau \in \{a, d, s\}$.*

(i) J_τ is the smallest set containing all t such that
  - *$\tau[t]$ is contained in a strong component of G_I which also contains variables $\tau[t_1], \tau[t_2]$ with $\tau[t_1] \geq c + \tau[t_2] \in I$ and $c \geq 1$; or*
  - *$\tau[t]$ is reachable in G_I from $\tau[t']$ with $t' \in J_\tau$.*

(ii) J_τ can be computed in linear time.

Proof. See [51], Theorem 2. □

By Proposition 39 we can sharpen the formulations of criteria (2) and (3) in Theorem 22. For criterion (1) we are only able to provide a more concrete form if $(\mu I)\ a[x'] \leq \infty$ for all pattern variables x'.

Corollary 40 *Given p and $I = \mathcal{I}(p)$.*

(1) *Assume $\mu I\ a[x'] < \infty$ for all pattern variables x'. Then $\mu I\ a[x] = \infty$ for some variable x iff some subterm $t_1\, t_2$ of p exists where t_2 contains a free variable $z \not\equiv t_2$ and $a[z]$ is in the same strong component of G_I as $a[t_2]$.*
(2) *$\mu I\, d[t'] = \infty$ for some t' iff some case-expression t in p exists with selector t_0 such that $d[t]$ is contained in the same strong component of G_I as $d[t_0]$.*
(3) *$\mu I\ s[t'] = \infty$ for some t' iff some subterm $t \equiv t_1\, t_2$ of p exists where $s[t_1]$ is contained in the same strong component of G_I as $s[t]$.*

Proof. For statement (1) assume that for every pattern variable x', $a[x']$ receives some value $< \infty$. Then every such $a[x']$ can only be contained in a strong component whose edges all correspond to constraints of the form $a[y] \geq a[y']$.

Now let $\mu I\ a[x] = \infty$ for some variable x. By Proposition 39, some strong component Q exists which contains an edge corresponding to a constraint $a[t_2] \geq c + a[z]$ with $c > 0$. Let z' be the variable for which $a[z'] \geq a[t_2] \in I$ such that there is a path from $a[z']$ to $a[z]$. Since z' cannot be a pattern variable, this constraint must have been generated for an application $t_1 t_2$ where $[t_1] \supseteq \lambda z'.t'$, i.e., $N[t_2] = pl \sqcup (c + a[z])$ for some polynomial pl. Especially, $a[z]$ is a free variable of $N[t_2]$. Hence, z must be a free variable of t_2 where $t_2 \not\equiv z$, and $a[t_2]$ is in the same strong component as $a[z]$. This gives us one direction of statement (1).

For the reverse direction assume $z \not\equiv t_2$ is a free variable of t_2, and $a[t_2]$ and

$a[z]$ are contained within the same strong component Q of the dependence graph G_I. Since $z \not\equiv t_2$, $N[t_2] = pl \sqcup (c + a[z])$ for some $c > 0$. It follows that Q contains an edge corresponding to constraint $a[t_2] \geq c + a[z]$ which, by Proposition 39, implies $\mu I\ a[z] = \infty$.

The characterizations of statements (2) and (3) directly follow from the observations that the $d[.]$-value is increased precisely when going from a case-expression to its selector, and that the $s[.]$-value is increased precisely when going from an application to its operator. □

In view of Corollary 40, three types of generalizations are sufficient to remove reasons for non-termination: generalization of the operator at an application, generalization of the argument at an application, and generalization of the selector at a case-expression. Specifically, we propose the following strategy for computing generalizations.

Algorithm 1 Given program p.

(a) Compute the set $I = \mathcal{I}(p)$;
(b) if μI is finite for all $a[t], d[t]$ and $s[t]$ then terminate.
(c) else generalize according to one of the following three rules:
  (1) – $t \equiv t_1\ t_0$ and t_0 contains a free variable $x \not\equiv t_0$ and $a[x]$ is in the same strong component of G_I as $a[t_0]$. Then generalize t_0 at t.
    – $t \equiv$ **case** t_0 **of** $q_1 \to t_1; \ldots; q_m \to t_m$ and for variable x of a pattern q_i, $\mu I\ a[x] = \infty$. Then generalize t_0 at t.
  (2) $t \equiv$ **case** t_0 **of** $q_1 \to t_1; \ldots; q_m \to t_m$ and $d[t]$ is contained in the same strong component of G_I as $d[t_0]$. Then generalize t_0 at t.
  (3) $t \equiv t_0\ t_1$ and $s[t_0]$ is contained in the same strong component of G_I as $s[t]$. Then generalize t_0 at t.
(d) goto (a). □

Note that generalizations never take place at let-expressions, individual variables, function names, constructor applications or lambda abstractions.

Our proposed strategy is non-deterministic. Termination of this strategy follows from Theorem 41 whereas correctness is the contents of Theorem 42.

Theorem 41 *Given program p, then at most $|p|$ generalizations are possible.*

Proof. Generalizations take place only at applications and case-expressions, and the number of each of these is not changed by any of the rules. Therefore, it suffices to verify that, if x is a let-bound variable, we do not generalize x

(i) at an application $t\,x$;

(ii) at a case-expression **case** x **of** $q_1 \to t_1; \ldots; q_m \to t_m$;
(iii) at an application $x\,t$.

Here (i) is true since we never generalize arguments that are variables.

For (ii), consider a case-expression $t \equiv \textbf{case}\ x\,\textbf{of}\,q_1 \to t_1; \ldots; q_m \to t_m$ where the selector x is a let-bound variable. Then the only simple constraints generated for pattern variables z of t are $[z] \supseteq \bullet$. This has two consequences.

First, no integer constraint with left-hand side $a[z]$, where z is such a pattern variable, is generated. This implies that $\mu I\ a[z] = 0$ for all of these.

The second consequence is that no integer constraint is generated whose right hand side contains $d[z]$ or $s[z]$, z a pattern variable of t. Hence, none of these $d[z]$ can be contained in a strong component containing at least one edge.

Therefore, (ii) is true as well.

Finally, assume we are given an application $x\,t$ where the operator x is a let-bound variable. Again, $[x] \supseteq \bullet$ is the only simple constraint generated for x. Therefore, $s[x]$ does not occur in the right-hand side of any integer constraint. This implies that $s[x]$ cannot be contained in a strong component with at least one edge. Consequently, no generalization according to rule (3) can be performed, showing that (iii) is true. □

The different types of generalization in the following theorem refer to Algorithm 1.

Theorem 42 *Given program p and $I = \mathcal{I}(p)$.*

(1) *If no generalization is possible according to (1) then $\mu I\ a[x] < \infty$ for all x.*
(2) *If no generalization is possible according to (2) then $\mu I\ d[t] < \infty$ for all t.*
(3) *If no generalization is possible according to (3) then $\mu I\ s[t] < \infty$ for all t.*

Proof. By Corollary 40. □

The following shows that deforestation with folding applied to a program computed by Algorithm 1 terminates.

Theorem 43 *Given program p. Algorithm 1 computes in time $\mathcal{O}(|p|^4)$ a program p' equivalent to p such that deforestation of p' encounters only finitely many different terms.*

Proof. By Theorems 22, 41, and 42. $\square$

The complexity $\mathcal{O}(|p|^4)$ of the preceding theorem may seem high. Factor $\mathcal{O}(|p|^3)$, however, originates from the control-flow component of the analysis called in every iteration whereas the remaining $|p|$ bounds the number of iterations. Of course, the crucial aspect of the algorithm mentioned in the preceding theorem is that the we introduce as few let-expression as possible. At the expense of possibly introducing unnecessary generalizations, the number of iterations can be reduced by generalizing more than one expression (per strong component) at a time.

Note that the output of deforestation is a higher-order treeless program with let-expressions; this means that the programs resulting from transformation construct no intermediate datastructures other than those that were retained for safety reasons.

9 Relation to higher-order treelessness

As mentioned in the introduction, Hamilton [23] and later Marlow [37] generalize the notion of treeless programs to the higher-order case. Their generalizations are slightly different, but in both cases treeless terms require arguments in applications and case-selectors to be variables. The following definition is Hamilton's version.

Definition 44 (Treeless programs) Let *treeless terms*, *functional terms*, and *treeless programs*, ranged over by tt, ft, and tp, respectively, be the subsets of general terms and programs defined by the grammar:

$$tt ::= x \mid c\, tt_1 \ldots tt_n \mid \textbf{case}\, x\, \textbf{of}\ q_1 \to tt_1; \ldots; q_k \to tt_k \mid \lambda x.\, tt \mid t\ x \mid f \mid \textbf{let}\ x = tt\ \textbf{in}\ tt'$$

$$ft ::= x \mid f \mid ft\, ft$$

$$tp ::= \textbf{letrec}\ f_1 = tt_1; \ldots; f_n = tt_n\ \textbf{in}\ \lambda x_1, \ldots, x_m.\, ft \qquad \square$$

Note that we do not demand treeless terms to be *linear*. In general, as can be seen by Example 15, deforestation is *not* guaranteed to terminate on treeless programs. Hamilton and Marlow therefore impose the additional restriction that programs be Hindley-Milner typable.

For simplicity we consider programs that are monomorphically typable. We assume that each variable has a specific type and consider simply typed λ-calculus á la Church [2] extended with inductive types and monomorphic recursion (see [39,28]). We write $t : \tau$ to express the fact that t has type τ.

Without loss of generality we may assume for a program $p \equiv \textbf{letrec}\ d_1; \ldots; d_n$ **in** t_{init} that all function names occurring in t_{init} are distinct and no function h is reachable from two distinct functions f_1, f_2 occurring in t_{init}. Any program can be brought to this form by suitable duplication of function definitions.

If p is monomorphically typable, then all values of $s[.]$-variables are finite. Therefore, in this case accumulating parameters and obstructing function calls are the only reasons for non-termination of deforestation. This is the contents of the next proposition.

Proposition 45 *Given a monomorphically typable program p, let $I = \mathcal{I}(p)$. Then $\mu I\ s[t] < \infty$ for all subterms t of p.*

Proof. First, we observe that our constraint system $\mathcal{C}(p)$ satisfies a *subject reduction* property: if $t : \tau$ and $[t] \supseteq t' \in \mathcal{C}(p)$ for some term t' then $t' : \tau$ as well.

Therefore, let D denote the set of all types of subterms of t_{init} and right hand sides of definitions in p and let $\leq$ be the reflexive and transitive closure of the smallest relation "$<$" satisfying $\tau < \tau \to \tau'$ for all τ, τ'.

Define function R on the subterms of p by: $R[t] = \tau$ iff $t : \tau$. Then:

(i) If $[t] \supseteq t' \in S$ then $R[t] = R[t']$.
(ii) If $t_1\ t_2$ is a subterm of p then $R[t_1] > R[t_1\ t_2]$.

These two properties imply that $\mu I\ s[t]$ is bounded above by the height of D, i.e., the maximal length of a strictly increasing chain in D. $\square$

Let us call monomorphic type τ *functional* iff τ is of the form $\tau_1 \to \tau_2$. Otherwise, we call τ *non-functional.* The following shows that given a monomorphically typable, higher-order treeless program, our analysis finds that no marks are required, provided all constructors have arguments of non-functional type only. Under the latter proviso this shows that our analysis is never worse than Hamilton's and Marlow's techniques. On the other hand, for many examples, our analysis is better.

Theorem 46 *Assume $p \equiv$ **letrec** $d_1; \ldots; d_n$ **in** t_{init} is monomorphically typable, higher-order treeless, and that all constructors in p have non-functional arguments only. Then conditions (1), (2) and (3) of Theorem 22 are all false.*

Proof. Let $C = \mathcal{C}(p)$ and $I = \mathcal{I}(p)$. Since p is monomorphically typable we know from Proposition 45 that $\mu I\ s[t] < \infty$ for all subterms t. It remains to prove that $\mu I\ a[x] < \infty$ and $\mu I\ d[t] < \infty$.

As in the proof of Proposition 45, we construct a finite partial ordering D together with ranking function R mapping the subterms t of p to elements in D. For this let us w.l.o.g. assume that $t_{init} \equiv \lambda z_1, \ldots, z_m . t_0$ where t_0 is of non-functional type. Then the carrier of D consists of all non-functional subterms occurring in t_0, ordered by the subterm ordering. Note that by assumption, t_0 is contained in D and is the maximal element.

Function R is now defined as follows. If t is a subterm of t_0 then $R[t]$ is the smallest superterm of t of non-functional type. Furthermore, if $R[f] = d$ then $R[t] = d$ and $R[x] = d$ for every subterm t and every bound variable x occurring in the right-hand side of f.

By assumption, all function names occurring in t_{init} are different and no function is reachable from two distinct functions in t_{init}. Therefore, R is well-defined.

Claim 1: Assume $t' \not\equiv \bullet$. Then the following holds:

Functional Type: If t has functional type, then
(i) $[t] \supseteq t'$ implies $R[t] = R[t']$.
(ii) If $t \equiv x$ and $a[x] \geq a[t'] \in I$ then t' is a variable or a subterm of t_0.
Non-functional Type: If t has non-functional type, then
(i) $[t] \supseteq t'$ implies $R[t] \geq R[t']$.
(ii) If $t \equiv x$ and $[x] \supseteq t'$ then t' is a variable or $R[x] > R[t']$.

Claim 2:

(i) If $a[t_1]$, $a[t_2]$ are in the same strong component of G_I then $R[t_1] = R[t_2]$.
(ii) If $d[t_1]$, $d[t_2]$ are in the same strong component of G_I then $R[t_1] = R[t_2]$.

First we infer Claim 2 from Claim 1. If $a[x] \geq a[t] \in I$ then also $[x] \supseteq t \in C$ and therefore by statements (i) of Claim 1, $R[x] \geq R[t]$. Since also $R[t] = R[z]$ for every variable occurring in t, the first assertion follows.

Now consider the $d[.]$-constraints. If $d[t_1] \geq d[t_2] \in I$ then also $[t_2] \supseteq t_1 \in C$ or t_1 is a superterm of t_2 with the same rank. Therefore again by statements (i) of Claim 1, $R[t_2] \geq R[t_1]$. If $d[t_1] \geq 1 + d[t_2] \in I$, then t_2 must be a case-expression with selector t_1. Since the rank of the selector of a case-statement equals the rank of the case-statement itself, the second assertion follows.

Next we show that Theorem 46 follows from Claims 1 and 2. For a contradiction assume that $a[t] \geq c + a[z] \in I$ such that $a[t]$ and $a[z]$ are in the same strong component Q and $c > 0$. Then in particular, z is a free variable of t but $t \not\equiv z$. Since both $a[t]$ and $a[z]$ are in Q, we find some $a[x] \in Q$, x a variable, such that also $a[x] \geq a[t] \in I$. Here t cannot be a subterm of t_0 since free variables z' of t_0 only receive values $\bullet$ implying that the corresponding $a[.]$-variables as well as $a[t]$ never occur in non-trivial strong components. Therefore, by statements (ii) of Claim 1, x must be of non-functional type with $R[x] > R[t]$—contradicting assertion (i) of Claim 2. We therefore conclude that $\mu I\ a[y] < \infty$ for all variables y.

Now assume for a contradiction, $d[t'] \geq 1 + d[t] \in I$ such that $d[t]$ and $d[t']$ are in the same strong component Q. Then t must be a case-expression with selector t'. Thus especially, $R[t] = R[t']$. By our syntactic assumptions, $t' \equiv x$ for some variable x of non-functional type. Since $d[x]$ as well as $d[t]$ are in the same strong component Q, some s exists such that $d[s] \in Q$ and $d[x]$ occurs in the right hand side of $d[s]$. Such constraints are only generated if also $[x] \supseteq s \in C$. If s is not a variable, then by Claim 1, $R[x] > R[s]$ in contradiction to Statement (ii) of Claim 2. If s is again a variable we may repeat this argument for $d[s]$ and the next variable $d[s']$ on a path from $d[x]$ to $d[t]$. We conclude that for no $t, t' \in Q$, $d[t'] \geq 1 + d[t] \in C$. Hence, $\mu I\ d[t] < \infty$ for all t.

It remains to prove Claim 1. The proof proceeds by induction on the number of iterations of the constraint computation. The assertion clearly holds for the *initial rules*. For the induction step, it suffices to make a case distinction on the *unfolding rules*. Therefore, assume $r[t] \sqsupseteq 1 \in C$.

Case 1: $t \equiv t_1\, t_2$ where $[t_1] \supseteq \lambda x.\, t' \in C$.

Then constraints $[t] \supseteq t'$, $[x] \supseteq t_2$ and $a[x] \geq a[t_2]$ are added to C. If t is of functional type, we get for the ranks:

$$\begin{aligned} R[t] &= R[t_1] && \text{(by definition)} \\ &= R[\lambda x.\, t'] && \text{(by induction)} \\ &= R[t'] && \text{(by definition)} \end{aligned}$$

If t is of non-functional type we have to replace the equality in the third line with "$\geq$". This gives assertion (i).

If x (and therefore also t_2) is of functional type, calculation of ranks gives:

$$\begin{aligned} R[x] &= R[\lambda x.\, t'] && \text{(by definition)} \\ &= R[t_1] && \text{(by induction)} \\ &= R[t] && \text{(by definition)} \\ &= R[t_2] && \text{(by definition)} \end{aligned}$$

If x is of non-functional type, we have to replace the equality in the fourth line with "$\geq$" according to statements (i).

Furthermore assume that t is a subterm of t_0. Then t_2 is a subterm of t_0 as well which conforms with assertion (ii) of the functional case. If, however, t_2 is additionally of non-functional type then $R[t_2] = t_2 < R[t] = R[x]$ as claimed by statement (ii) of the non-functional case.

If on the other hand, t is *not* a subterm of t_0, then t is a subterm of a right-hand side of some definition. Since p is treeless, this implies that t_2 is necessarily a variable—giving statements (ii) also in this case.

Case 2: $t \equiv \textbf{let}\ x{=}t_1\ \textbf{in}\ t_2$.

Then constraints $[t] \supseteq t_2$ and $[p] \supseteq t_1$ are generated. Since by definition, $R[t_2] = R[t]$ the assertion follows.

Case 3: $t \equiv \textbf{case}\ z\ \textbf{of}\ \ q_1 \to t_1; \ldots; q_m \to t_m$.

By definition, $R[t] = R[t_j]$ for all j. Therefore, statements (i) hold for all generated constraints $[t] \supseteq t_j$.

Now assume x equals pattern variable x_i in pattern $q_j \equiv c\, x_1 \ldots x_n$ of the case-expression where $[z] \supseteq c\, s_1 \ldots s_n$. Then constraint $[x] \supseteq t'$ is generated with $t' \equiv s_i$. By our restriction on the types of constructors, x must be of non-functional type. Then we know:

$$
\begin{aligned}
R[x] &= R[t] && \text{(by definition)} \\
&= R[z] && \text{(by definition)} \\
&> R[c\, s_1 \ldots s_n] && \text{(by induction)} \\
&\geq R[t'] && \text{(by definition)}
\end{aligned}
$$

in accordance to statement (ii) of the non-functional case.

This concludes the proof of Claim 1 and also the proof of Theorem 46. Note that for our proof, we relied on monomorphic typability of the program in two essential ways: first, in order to avoid accumulating spines, and second, to classify terms. In a polymorphic setting, the same term may be used with both a functional and a non-functional type. □

The restriction that constructors may not have functional arguments is a weakness of our analysis in its present form.

Example 47 Consider the following program.

$$\begin{array}{l}\textbf{letrec}\\ \quad I = \lambda z.\, \textbf{case}\ z\ \textbf{of}\ [\,] \to [\,];\ (h : t) \to h : I\ t\\ \textbf{in}\ \lambda x.\, I\ (I\ x)\end{array}$$

Unfolding the outer call to I in term $I\ (I\ x)$ leads to the case-expression $\textbf{case}\ (I\ x)\ \textbf{of}\ [\,] \to [\,];\ (h : t) \to h : I\ t$ in which the inner call to I must be unfolded. Superficially, a call to I in the empty context leads to a new call to I in a non-empty context, with the risk of deforestation proceeding indefinitely. The truth is that the two calls to I are unrelated, and the problem could be solved by considering instead the following program:

$$\begin{array}{l}\textbf{letrec}\\ \quad I_1 = \lambda z_1.\, \textbf{case}\ z_1\ \textbf{of}\ [\,] \to [\,];\ (h_1 : t_1) \to h_1 : I_1\ t_1\\ \quad I_2 = \lambda z_2.\, \textbf{case}\ z_2\ \textbf{of}\ [\,] \to [\,];\ (h_2 : t_2) \to h_2 : I_2\ t_2\\ \textbf{in}\ \lambda x.\, I_1\ (I_2\ x)\end{array}$$

In the first-order case this trick is sufficient to ensure that no generalizations are performed on treeless programs [59]. However, in the higher-order case, the problematic situation may arise after a number of transformation steps as in the program:

$$\begin{array}{l}\textbf{letrec}\\ \quad I\ = \lambda z.\, \textbf{case}\ z\ \textbf{of}\ [\,] \to [\,];\ (h : t) \to h : I\ t\\ \quad G = \lambda d.\, \textbf{case}\ d\ \textbf{of}\ (c\ h\ a) \to h\ a\\ \quad H = \lambda f, y.\, c\ f\ (f\ y)\\ \textbf{in}\ \lambda x.\, G\ (H\ I\ x)\end{array}$$

The restriction on treeless programs that constructors may not have functional arguments is sufficient to prevent this problem.

There are two reasons why the restriction may not be serious: first, it is not clear how often programs actually make use of constructors with functional arguments; and second, it is only in some special cases that our analysis is confused by such constructors. $\square$

Another weakness in our analysis stems from the fact that there is an explicit bound on the finite values of variables in systems of integer constraints. More specifically, for some program p and $I = \mathcal{I}(p)$, if $\mu I\ \tau[t] < \infty$ then $\mu I\ \tau[t] < |p|$.

This implies that our analysis might be overly conservative on programs for which transformation unfolds in a context whose depth exceeds $|p|$.

An investigation of possible enhancements of our analysis to avoid occasional deficiencies of this type remains for future work. Especially, an enhancement of precision in the control-flow component along the lines suggested by Hanne Riis Nielson and Flemming Nielson in [41] seems promising.

10 Other related work

The formulations of deforestation considered so far may all be described as *interpretive*; the deforestation algorithm is essentially an interpreter modified so as to take terms with free variables into account. As a consequence, deforestation applied to a program may not terminate. Even applied to safely blazed programs, the blazed deforestation algorithm must maintain a list of all terms it has encountered and fold when possible, a possibly costly technique.

Another approach, which might be called *algebraic*, deals with programs in fixed recursion schemes, and uses algebraic laws to transform programs, e.g., Wadler's Theorems for free [67]. Several lines of work use this approach.

Gill, Launchbury, and Peyton-Jones [19] remove intermediate lists explicitly produced and consumed by means of the primitives `build` and `foldr` within the same function. No unfolding or folding is required. The approach is simple and cheap, but less general because it only applies to explicit production and consumption of intermediate lists by means of `build` and `foldr` within the same function. However, this is not as bad as it sounds. First, many library functions can be written in terms of `build` and `foldr`, enabling optimizations on programs using these library functions. Second, some general function definitions can be automatically converted into the `build-foldr` form, as mentioned below. Finally, the technique can be generalized from lists to other data structures, as is also mentioned below. In a subsequent paper [20], it turned out that the technique needed additional transformation steps in order to give good results on many examples. This is elaborated in more detail in Gill's thesis [18], which also gives numerous practical experiments with an actual implementation of the technique in a Haskell compiler.

Sheard and Fegaras [55] independently invented a related technique which is not limited to lists, and extended it so as to apply to functions defined inductively on several arguments [16], like the `zip` function. As Gill [18] points out, the `build-foldr` technique did not work well on such functions. Later, Launchbury and Sheard [33] show how some classes of functions can be transformed automatically into the form that explicitly produces and consumes data structures.

Takano and Meijer [61] extend these algebraic techniques to more general data types and recursion schemes. Their method is capable of some optimization that the `build-foldr` technique is incapable of, even when both are applied to programs constructing intermediate lists.

As mentioned, the algebraic approaches rely on functions being written in explicit recursion patterns, although some functions can be transformed into this form automatically. Hence, there are cases where interpretive deforestation applies and algebraic approaches fail. However, the opposite is also the case. Gill [18] gives one example, and Sheard and Fegaras [56] show that a variant of their technique can obtain effects like *tupling* which are beyond deforestation. See also [37].

We end the paper by noting that introducing a technique to ensure termination of deforestation requires some motivation, since there are several transformers with similar aims and effects as deforestation with known techniques for ensuring termination. Most notably, termination techniques exist for *positive supercompilation* [60], *partial evaluation* [30], and *partial deduction* [34]. The techniques for positive supercompilation and partial deduction are *online*; that is, the efforts taken to ensure termination are carried out *during* the transformation process. In contrast, the technique for deforestation described in this paper is *offline*; it is applied *before* the transformation process. We believe that offline techniques are to be preferred, since such techniques often lead to shorter transformation time. The technique for partial evaluation mentioned above is offline, but relies crucially on the fact that the partial evaluator follows a *call-by-value* transformation strategy; that is, arguments of a function call are transformed before the function call itself. This is not the case in deforestation, and therefore the termination problem here is somewhat harder, see [40].

Acknowledgement

We are indebted for discussions on higher-order deforestation to Geoff Hamilton and Wei-Ngan Chin.

References

[1] Z.M. Ariola, M. Felleisen, M. Maraist, J. Odersky, and P. Wadler. A call-by-need lambda-calculus. In *Conference Record of the Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, pages 233–246. ACM Press, 1995.

[2] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–309. Oxford University Press, 1992.

[3] R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[4] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, Department of Computer Science, Universty of Copenhagen, 1990. DIKU-Rapport 90/17.

[5] A. Bondorf. Automatic autoprojection of recursive equations. *Science of Computer Programming*, 17(1-3):3–34, 1991.

[6] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machines*, 24(1):44–67, 1977.

[7] W.-N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, 1990.

[8] W.-N. Chin. Generalising deforestation to all first-order functional programs. In *Workshop on Static Analysis of Equational, Functional and Logic Programming Languages, BIGRE 74*, pages 173–181, 1991.

[9] W.-N. Chin. Fully lazy higher-order removal. In *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 38–47, 1992. Yale University technical report YALEU/DCS/RR-909.

[10] W.-N. Chin. Safe fusion of functional expressions. In *ACM Conference on Lisp and Functional Programming*, pages 11–20. ACM Press, 1992.

[11] W.-N. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994.

[12] W.-N. Chin and J. Darlington. Higher-order removal transformation technique for functional programs. In *Australian Computer Science Conference*, volume 14,1 of *Australian CS Comm.*, pages 181–194, 1992.

[13] W.-N. Chin and S.-C. Khoo. Better consumers for deforestation. In D.S. Swierstra, editor, *Programming Languages: Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 223–240. Springer-Verlag, 1995.

[14] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46, 1981.

[15] M.S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, 1982.

[16] L. Fegaras, T. Sheard, and T. Zhou. Improving programs which recurse over multiple inductive structures. In *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1994.

[17] A. Ferguson and P.L. Wadler. When will deforestation stop? In *Glasgow Workshop on Functional Programming*, pages 39–56, 1988.

[18] A.J. Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Department of Computing Science, Glasgow University, 1996.

[19] A.J. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming and Computer Architecture*, pages 223–232. ACM Press, 1993.

[20] A.J. Gill and Simon L. Peyton Jones. Cheap deforestation in practice: An optimiser for Haskell. In *IFIP*, pages 581–586, 1994.

[21] G. Hamilton. *Compile-Time Optimisations of Storage Usage in Lazy Functional Programs.* PhD thesis, University of Stirling, 1993.

[22] G. Hamilton. Extending first order deforestation. Technical Report TR 95-06, Department of Computer Science, Keele University, 1995.

[23] G. Hamilton. Higher order deforestation. Technical Report TR 95-07, Department of Computer Science, Keele University, 1995.

[24] G. Hamilton. Higher order deforestation. In H. Kuchen and S.D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 213–227. Springer-Verlag, 1996.

[25] G. Hamilton and S.B. Jones. Extending deforestation for first order functional programs. In *Glasgow Workshop on Functional Programming*, pages 134–145, 1991.

[26] G. Hamilton and S.B. Jones. Transforming programs to eliminate intermediate structures. In *Workshop on Static Analysis of Equational, Functional and Logic Programming Languages, BIGRE 74*, pages 182–188, 1991.

[27] N. Heintze. Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[28] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[29] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[30] C.K. Holst. Finiteness analysis. In J. Hughes, editor, *Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 495–519. Springer-Verlag, 1991.

[31] J. Hughes. Why functional programming matters. In D. Turner, editor, *Research Topics in Functional Programming.* Addison-Wesley, 1990.

[32] R.B. Kierburtz and J. Schultis. Transformations of FP program schemes. In *Conference on Functional Programming and Computer Architecture*, pages 41–48. ACM Press, 1981.

[33] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conference on Functional Programming and Computer Architecture*, pages 314–323. ACM Press, 1995.

[34] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation.*, volume 1110 of *Lecture Notes in Computer Science*, pages 263–283. Springer-Verlag, 1996.

[35] Z. Manna and R. Waldinger. Synthesis: Dreams => programs. *IEEE Transations on Software Engineering*, 5(4):157–164, 1979.

[36] S. Marlow and P.L. Wadler. Deforestation for higher-order functions. In J. Launchbury, editor, *Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 154–165, 1992.

[37] S.D. Marlow. *Deforestation for Higher-Order Functional Languages.* PhD thesis, University of Glasgow, 1996.

[38] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[39] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th International Conference on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

[40] K. Nielsen and M.H. Sørensen. Call-by-name CPS-translation as a binding-time improvement. In A. Mycroft, editor, *Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 296–313. Springer-Verlag, 1995.

[41] H.R. Nielson and F. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345. ACM Press, 1997.

[42] J. Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17:47–82, 1995.

[43] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17:576–599, 1995.

[44] D. Sands. Proving the correctness of recursion-based automatic program transformation. In P. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 681–695. Springer-Verlag, 1995.

[45] D. Sands. Total correctness by local improvement in program transformation. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 221–232. ACM Press, 1995.

[46] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, A(167), 1996.

[47] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2), March 1996.

[48] W.L. Scherlis. Program inprovement by internal specialization. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 41–49. ACM Press, 1981.

[49] H. Seidl. Least solutions of equations over $\mathcal{N}$. In *International Colloquium on Automata, Languages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 400–411, 1994.

[50] H. Seidl. Integer constraints to stop deforestation. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 326–340. Springer-Verlag, 1996.

[51] H. Seidl. Least and greatest solutions of equations over $\mathcal{N}$. *Nordic Journal of Computing*, 3:41–62, 1996.

[52] H. Seidl and M.H. Sørensen. Constraints to stop higher-order deforestation. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 400–413. ACM Press, 1997.

[53] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506, Amsterdam, 1988. North-Holland.

[54] P. Sestoft. Replacing function parameters by global variables. In *Conference on Functional Programming and Computer Architecture*, pages 29–53. ACM Press and Addison-Wesley, 1989.

[55] T. Sheard and L. Fegaras. A fold for all seasons. In *Conference on Functional Programming and Computer Architecture*, pages 233–242. ACM Press, 1993.

[56] T. Sheard and L. Fegaras. Optimizing algebraic programs. Technical Report 94-004, Oregon Graduate Institute, 1994.

[57] O. Shivers. Control-flow analysis in scheme. In *Programming Languages: Design and Implementation*, pages 114–128, 1988.

[58] O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[59] M.H. Sørensen. A grammar-based data-flow analysis to stop deforestation. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, 1994.

[60] M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.

[61] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conference on Functional Programming and Computer Architecture*, pages 306–313. ACM Press, 1995.

[62] V.F. Turchin, R. Nirenberg, and D. Turchin. Experiments with a supercompiler. In *ACM Conference on Lisp and Functional Programming*, pages 47–55. ACM Press, 1982.

[63] P.L. Wadler. Applicative style programming, program transformation, and list operators. In *Conference on Functional Programming and Computer Architecture*, pages 25–32. ACM Press, 1981.

[64] P.L. Wadler. Listlessness is better than laziness. In *ACM Conference on Lisp and Functional Programming*, pages 282–305. ACM Press, 1984.

[65] P.L. Wadler. Listlessness is better than lazyness II: Composing listless functions. In *Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[66] P.L. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[67] P.L. Wadler. Theorems for free! In *Conference on Functional Programming and Computer Architecture*. ACM Press, 1989.

[68] P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.