Constraints to Stop Higher-Order 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 removes intermediate data structures from functional programs. To be appropriate 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 a technique for higher-order programs was only recently introduced by Hamilton and later 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.

1 Introduction

Lazy, higher-order, functional programming languages lend themselves to a certain style of programming which uses intermediate data structures [28]. However, this style also leads to inefficient programs.

Example 1 Consider the following program.

$$\begin{array}{l}\textbf{letrec}\\ \quad a \;=\; \lambda x, y.\, \textbf{case}\ x\ \textbf{of}\\ \qquad\qquad\qquad [] \;\to\; y\\ \qquad\qquad\quad (h:t) \to\; h : a\ t\ y\\ \textbf{in}\ \lambda u, v, w.\ a\ (a\ u\ v)\ w\end{array}$$

The main term $\lambda u, v, w.\, a\,(a\,u\,v)\,w$ appends the three lists u, v, and w. Appending u and v results in an intermediate list to which w is appended. Allocation and deallocation of the intermediate list at run-time is expensive. Sacrificing clarity for efficiency, we would prefer a program such as:

$$\begin{array}{l}\textbf{letrec}\\ \quad da \;=\; \lambda x, y, z.\, \textbf{case}\ x\ \textbf{of}\\ \qquad\qquad\qquad [] \;\to\; a'\ y\ z\\ \qquad\qquad\quad (h:t) \to\; h : da\ t\ y\ z\\ \quad a' \;=\; \lambda y, z.\, \textbf{case}\ y\ \textbf{of}\\ \qquad\qquad\qquad [] \;\to\; z\\ \qquad\qquad\quad (h:t) \to\; h : a'\ t\ z\\ \textbf{in}\ \lambda u, v, w.\ da\ u\ v\ w\end{array}$$

For instance, 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. More substantial examples appear in [34]. □

Ideally we should write the first version, and have it translated to the second automatically, e.g., by our compiler. This is indeed done by Wadler's [16, 53, 54] deforestation which eliminates intermediate data structures from first-order functional programs.[1] Deforestation terminates on *treeless* programs. Subsequent techniques to ensure termination of deforestation on all first-order programs are due to Chin [6, 7, 9, 10, 12], and later to Hamilton [20, 21, 23, 24]. The essence of these techniques is to annotate all parts of the program that violate the treeless syntax, and then let the deforestation algorithm skip over annotated parts.

Following a suggestion of Jones, the second author [47] developed a technique that annotates fewer parts of the program. Inspired by earlier work on *tree-grammars* by Jones [29], the technique computes a tree-grammar which approximates the set of all terms encountered by deforestation of some program. Inspired by work of Heintze [25], the first author [44] reformulated this technique in terms of *set constraints* and observed that the information necessary to detect dangerous subterms can be computed without explicitly building the approximating system of set constraints. Instead, it suffices to perform a control-flow analysis in the sense of Palsberg [37] and Palsberg and O'Keefe [38] and, while doing so, to collect a set of *integer constraints*.

These termination techniques concern only *first-order* programs. However, modern functional languages like ML, Haskell, and Miranda include higher-order functions which should be transformed too. Several preliminary approaches

[1] Earlier techniques include [3, 5, 13, 14, 30, 32, 49, 50, 51, 52].

reduce the higher-order case to the first-order case. Wadler [54] considers programs with *higher-order macros*. Any such program typable in the Hindley-Milner [27, 35] 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 [6, 7, 9, 10] starts out with a higher-order program and uses a higher-order removal transformation [6, 8, 11] 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 annotated programs in the restricted higher-order form, and annotates any remaining higher-order parts as well as first-order parts violating the treeless syntax. In the process of applying deforestation to such a program, higher-order subterms may reappear, and these are again removed by the higher-order removal algorithm during deforestation. The 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 [33], who leave open the question of guaranteeing termination. This was addressed by Hamilton [22], who gives a formulation of the higher-order deforestation algorithm applicable to annotated programs and introduces a notion of higher-order treelessness. He then proves that deforestation of any Hindley-Milner typable program terminates, if all parts of the program violating the higher-order treeless syntax are annotated. Inspired by Hamilton's work, Marlow re-evaluated his earlier research [34]. He gave a similar notion of higher-order treelessness and proved that transformation of any Hindley-Milner typable higher-order program terminates, if all parts of the program violating the higher-order treeless syntax are annotated. Marlow has implemented this 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 annotating and thereby skipping over parts of programs that could have been improved.

Example 2 Consider the following program.

$$
\begin{array}{l}
\textbf{letrec} \\
\quad \begin{array}{lcl}
c & = & \lambda x, xs.x : xs \\
foldr & = & \lambda f, a, l.\,\textbf{case}\ l\ \textbf{of} \\
 & & \qquad [] \ \rightarrow\ a \\
 & & \qquad (z : zs) \rightarrow\ f\ z\ (foldr\ f\ a\ zs)
\end{array} \\
\textbf{in}\ \lambda u, v, w.foldr\ c\ w\ (foldr\ c\ v\ u)
\end{array}
$$

The term $foldr\,c\,v\,u$ is a higher-order formulation of the term $a\,u\,v$ from Example 1. The whole program is therefore equivalent to the program in Example 1, and we would expect to be able to transform it into the more efficient program in Example 1. This is indeed what happens when we apply deforestation to the program. However, the techniques by Hamilton and Marlow require that the argument $foldr\,f\,a\,zs$ in the definition of *foldr* be annotated, and this prevents the desired transformation. □

There are many such examples. Chin [10] 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 annotation scheme is purely syntactic; it does not take into account what actually happens during deforestation.

In this paper we give a new technique to ensure termination of higher-order deforestation. We adopt a version of Hamilton's higher-order deforestation algorithm applicable to annotated terms, but do not annotate all parts violating the higher-order treeless syntax. Before transformation we instead compute a set of constraints approximating the set of terms encountered during deforestation of the program. This can be done efficiently using well-known techniques. While doing so, we extract quantitative information to detect whether deforestation will proceed indefinitely, and if so, we annotate parts of the program responsible for the indefinite transformation. The technique is a generalization of our technique for first-order deforestation [47, 44].

Section 2 presents our higher-order language, and Section 3 presents higher-order deforestation. Section 4 shows the sources of non-termination of deforestation. Section 5 introduces constraint systems, and Section 6 uses constraints to approximate deforestation. Section 7 shows how to calculate annotations that ensure termination of deforestation, from the set of approximating constraints. Section 8 relates the approach to that by Hamilton and Marlow. Section 9 concludes. Proof sketches are given in the appendices.

2 Language and notation

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 grammar:

$$
\begin{array}{lcl}
t & ::= & x \mid \lambda x.t \mid c\ t_1 \ldots t_n \mid f \mid \textbf{let}\ v{=}t\ \textbf{in}\ t' \mid t\ t' \mid \\
 & & \textbf{case}\ t_0\ \textbf{of}\ q_1{\rightarrow}t_1; \ldots; q_k{\rightarrow}t_k \\
q & ::= & c\ x_1 \ldots x_n \\
d & ::= & f = t \\
p & ::= & \textbf{letrec}\ d_1; \ldots; d_n\ \textbf{in}\ t
\end{array}
$$

(where $n \geq 0, k > 0$). The t_0 in **case**-expressions is called the *selector*. In applications t is the *operator* and t' the *argument*. Programs must be *closed*, i.e., all variables of t in definitions $f = t$ and in programs $\textbf{letrec}\ d_1; \ldots; d_n\ \textbf{in}\ t$ must be bound. No variable may occur more than once in a pattern. To each function call must correspond exactly one definition, and the patterns in a **case**-expression must be non-overlapping and exhaustive. We assume that terms of form $(c\,t_1 \ldots t_n)\,t$ and $\textbf{case}\ (\lambda x.t)\,\textbf{of}\,q_1{\rightarrow}t_1; \ldots; q_n{\rightarrow}t_n$ never arise. The semantics of the language is call-by-need [1].

$\mathrm{FV}(t)$ 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. Variable names in the input program are assumed to be unique. We 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 []. We also write $[x_1, \ldots, x_n]$ for $x_1 : \ldots : x_n : []$. Substitution of t' for x in t is written $t\{x := t'\}$. □

As in [33], the **let**-construct is used instead of annotations. Instead of annotating dangerous parts of a program and having deforestation work conservatively on annotated subterms, we transform dangerous parts of the program into **let**-expressions and let deforestation work conservatively on **let**-expressions. This yields less syntactic overhead than working with annotations.

3 The higher-order deforestation algorithm

To formulate the deforestation algorithm 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 simulates call-by-name evaluation, so there is always a unique subterm whose reduction is forced. For instance, to find out which branch to choose in

$$\textbf{case } f\ t \textbf{ of } [] \to []; (x : xs) \to x : a\ xs\ ys$$

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.,

$$\textbf{case } \langle\rangle\ t \textbf{ of } [] \to []; (x : xs) \to x : a\ xs\ ys$$

is the *context*.

Definition 4 Let e, r, o range over contexts, redexes, and observables, respectively, as defined by the grammar:

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

Let $e\langle t\rangle$ denote the result of replacing $\langle\rangle$ in e by t. □

Every term t is an observable or decomposes uniquely into a context e and redex r with $t \equiv e\langle r\rangle$. Stating that $t \equiv e\langle r\rangle$ does not mean that t has any brackets—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.

In the following definition, inspired by [22], the clauses are mutually exclusive and together exhaustive. The essence of the rules is that, 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 $e\langle r'\rangle$.

Definition 5 (Deforestation)

$$[\![\textbf{letrec } d_1; \ldots; d_n \textbf{ in } t_{init}]\!] = [\![t_{init}]\!] \quad (0)$$

$$[\![x\ t_1 \ldots t_n]\!] = x\ [\![t_1]\!] \ldots [\![t_n]\!] \quad (1)$$

$$[\![c\ t_1 \ldots t_n]\!] = c\ [\![t_1]\!] \ldots [\![t_n]\!] \quad (2)$$

$$[\![\lambda x.t]\!] = \lambda x.[\![t]\!] \quad (3)$$

$$[\![e\langle f\rangle]\!] = [\![e\langle t^f\rangle]\!] \quad (f = t^f) \quad (4)$$

$$[\![e\langle(\lambda x.t)\ t'\rangle]\!] = [\![e\langle t\{x := t'\}\rangle]\!] \quad (5)$$

$$[\![e\langle\textbf{let } x{=}t \textbf{ in } t'\rangle]\!] = \textbf{let } x{=}[\![t]\!] \textbf{ in } [\![e\langle t'\rangle]\!] \quad (6)$$

$$[\![e\langle\textbf{case } x\ t_1 \ldots t_n \textbf{ of } q_1 \to s_1; \ldots; q_k \to s_k\rangle]\!] = \textbf{case } x\ [\![t_1]\!] \ldots [\![t_n]\!] \textbf{ of } q_1 \to [\![e\langle s_1\rangle]\!]; \ldots; q_k \to [\![e\langle s_k\rangle]\!] \quad (7)$$

$$[\![e\langle\textbf{case } c\ t_1 \ldots t_n \textbf{ of } q_1 \to s_1; \ldots; q_k \to s_k\rangle]\!] = [\![e\langle s_j\{x_1 := t_1, \ldots, x_n := t_n\}\rangle]\!] \quad (q_j \equiv c\ x_1\ \ldots x_n) \quad (8)$$

Definitions $f = t^f$ in (4) are taken from $d_1, \ldots, d_n$. □

As is well-known, Algorithm 5 hardly ever terminates. For instance, on program $\textbf{letrec } f = f \textbf{ in } f$ the 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 6 (Folding) Let $[\![.]\!]$ take a parameter I. In clause (1-3,5-8) of Definition 5, I is passed unchanged to the recursive calls of $[\![.]\!]$. Replace (0,4) by:

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

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

where $\mathrm{FV}(e\langle f\rangle) = \{x_1, \ldots, x_n\}$ in some canonical 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. □

Example 7 We now show how deforestation transforms the first program in Example 1 into the second more efficient one. For brevity we adopt the abbreviations:

$$\begin{array}{l} I = \{da = \lambda u,v,w.a\ (a\ u\ v)\ w\} \\ I' = \{f = \lambda u,v,w.\textbf{case } (a\ u\ v) \textbf{ of } [] \to w; (h{:}t) \to h : a\ t\ w\} \\ I'' = \{a' = \lambda y,z.a\ y\ z\} \end{array}$$

Then transformation proceeds as follows.

$$\begin{array}{ll} & [\![\lambda u, v, w.\ a\ (a\ u\ v)\ w]\!]\{\} \\ & \qquad (3) \\ = & \lambda u, v, w.\ [\![a\ (a\ u\ v)\ w]\!]\{\} \\ & \qquad (4') \\ = & \lambda u, v, w.\ da\ u\ v\ w \\ & \qquad \textbf{where} \\ da = & \lambda u, v, w.[\![(\lambda x, y.\textbf{case } x \textbf{ of} \\ & \qquad [] \to w \\ & \qquad (h : t) \to h : a\ t\ y)\ (a\ u\ v)\ w]\!]I \\ & \qquad (5) \\ = & \lambda u, v, w.[\![\textbf{case } (a\ u\ v) \textbf{ of} \\ & \qquad [] \to w \\ & \qquad (h : t) \to h : a\ t\ w]\!]I \\ & \qquad (4') \\ = & \lambda u, v, w.f\ u\ v\ w \\ & \qquad \textbf{where} \\ f = & \lambda u, v, w.[\![\textbf{case } ((\lambda x, y.\textbf{case } x \textbf{ of} \\ & \qquad [] \to y \\ & \qquad (g : s) \to g : a\ s\ y)\ u\ v) \textbf{ of} \\ & \qquad [] \to w \\ & \qquad (h : t) \to h : a\ t\ w]\!]I \cup I' \\ & \qquad (1, 2, 5, 7) \\ = & \lambda u, v, w.\textbf{case } u \textbf{ of} \\ & \qquad [] \to \textbf{case } v \textbf{ of} \\ & \qquad\qquad [] \to w \\ & \qquad\qquad (h : t) \to h : [\![a\ t\ w]\!]I \cup I' \\ & \qquad (h' : t') \to h' : [\![a\ (a\ t'\ v)\ w]\!]I \cup I' \\ & \qquad (4') \\ = & \lambda u, v, w.\textbf{case } u \textbf{ of} \\ & \qquad [] \to \textbf{case } v \textbf{ of} \\ & \qquad\qquad [] \to w \\ & \qquad\qquad (h : t) \to h : a'\ t\ w \\ & \qquad (h' : t') \to h' : da\ t'\ v\ w \\ & \qquad \textbf{where} \\ a' = & \lambda t, w.[\![\textbf{case } t \textbf{ of} \\ & \qquad [] \to w \\ & \qquad (h' : t') \to h' : a\ t'\ w]\!]I \cup I' \cup I'' \\ & \qquad (1, 2, 4', 7) \\ = & \lambda t, w.\textbf{case } t \textbf{ of} \\ & \qquad [] \to w \\ & \qquad (h' : t') \to h' : a'\ t'\ w \end{array}$$

Hence the new program is

$$
\begin{array}{l}
\textbf{letrec} \\
\quad da \;=\; \lambda x, y, z.f\ u\ v\ w \\
\quad f \;=\; \lambda x, y, z.\,\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad [] \;\to\; \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad [] \;\to\; z \\
\qquad\qquad\qquad\qquad\qquad (h:t) \to\; h : a'\ t\ z \\
\qquad\qquad\quad (h:t) \to\; h : da\ t\ y\ z \\
\quad a' \;=\; \lambda y, z.\,\textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad [] \;\to\; z \\
\qquad\qquad\quad (h:t) \to\; h : a'\ t\ z \\
\textbf{in}\ \lambda u, v, w.da\ u\ v\ w
\end{array}
$$

This is equivalent to the efficient program in Example 1. Unnecessary auxiliary functions, like f above, can easily be unfolded in a post-processing phase. □

Definition 8 (Encountered terms) Given 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 9 In Example 7, the terms $\lambda u, v, w.a\,(a\,u\,v)\,w$ and $\lambda u, v, w.(\lambda x, y.\textbf{case}\ x\ \textbf{of}\ [] \to w; (h:t) \to h : a\ t\ y)\,(a\,u\,v)\,w$ are the first two terms encountered by deforestation. □

Remark 10 Strictly speaking, in Definition 6 we should have replaced clause (5) by a clause (5') analogous to (4'). The resulting algorithm can be proved to terminate whenever the one in Definition 5 encounters only finitely many different terms. However, in a number of situations, e.g. when the program is either linear or typed, this is not necessary. In what remains we simply assume that there is some way of folding such that if the algorithm in Definition 5 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 5 encounters only finitely many different terms. □

Apart from termination, there are two other aspects of correctness for deforestation: preservation of operational semantics and non-degradation of efficiency. A proper development of these two aspects is beyond the scope of this paper, so we end this section with a brief review. 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.

As for preservation of operational semantics, the output of deforestation should be equivalent to the input. That each step of deforestation preserves call-by-need semantics is easily proved, but extending the proof to account for folding is more involved. A general technique due to Sands [40, 42] can be used to prove this [39, 41].

As for non-degradation in efficiency, the output of deforestation should be at least as efficient as the input. First, there is the problem of avoiding *duplication of computation.* Transformation can change a polynomial time program into an exponential time program. This can be avoided by considering only programs consisting of functions that do not duplicate their arguments [54]. Weaker restrictions are also known [45, 4, 20]. Second, there is the problem of *code duplication.* Unrestrained unfolding may increase the size of a program dramatically. In principle this increase does not affect running-time, but in practice it may. Third, transformation steps can *loose laziness* and *full laziness*, see [34].

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 show that deforestation of certain programs loops indefinitely, but with certain changes in the programs, deforestation terminates. These changes are called *generalizations.*

Example 11 (The Accumulating Parameter) Consider the following program:

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

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 the formal 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:

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

Deforestation applied to this program terminates. □

Example 12 (The Obstructing Function Call) Now consider the program:

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

The r function again reverses its argument, first reversing the tail and then appending the head at the end. Deforestation encounters the terms r, and $\lambda zs_1, z_1.$ **case** $(r zs_1)$ **of** $\cdots$, and $\lambda zs_2, z_2, z_1.$ **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:

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

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

Example 13 (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, e.g., simple types. 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:

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

Then deforestation terminates. □

The operation of going from the term $t\{x := t'\}$ to $\textbf{let}\ x = t'\ \textbf{in}\ t$ is called *generalization of* t' *at* $t\{x := t'\}$. In Example 11, 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 12, we generalized the call $r\ xs$ at the **case**-expression in the body of the definition of r. In example 13, we generalized f at the application $f\ x$ in the body of the definition of f.

Generalizing should be thought of as annotating. Instead of putting funny symbols on our programs we introduce **let**-expressions.

5 Constraint Systems

This section introduces the necessary theory regarding constraint systems.

Let D be a complete lattice. For some variable set $\mathcal{V}$, we consider sets S of constraints of the form

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

where $X, X_1, \ldots, X_n \in \mathcal{V}$, and f denotes a monotonous function $[\![f]\!] : D^n \to D$. Then S has a least model μS mapping variables to elements of D such that

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

for every constraint $X \sqsupseteq f X_1 \ldots X_n \in S$. We shall make use of two instances of this sort of constraints: *simple constraints* and *integer constraints*.

In a set of simple constraints, a finite set A of objects is given. D is the power-set of A ordered by set inclusion. In our application we need no occurrences of variables or operators in right-hand sides. So, they are of the simple form $X \supseteq a$ for some $a \in A$.[2] 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 the complete lattice D is the non-negative numbers $\mathcal{N}$ equipped with their natural ordering and extended by ∞. Right hand sides are built up from variables and constants by means of certain operators, in our case "+" and "$\sqcup$" (maximum).

Example 14 Consider the integer constraints:

$$
\begin{array}{lcllcl}
X & \geq & 1 & \qquad Z & \geq & X + Y \\
Y & \geq & 7 & \qquad Y & \geq & Z \sqcup X
\end{array}
$$

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

Since $\mathcal{N}$ does not satisfy the ascending chain condition, naive fix-point iteration may not suffice to compute least models. The first author [43, 44] presents algorithms that do compute such least models.[3] For the systems we consider, least models can be computed in linear time.

6 Approximating deforestation

We now present an analysis which computes, for a given program, a set of integer constraints whose least model indicates which subterms cause termination problems. In the next section we show how to compute generalizations from such constraint systems.

The analysis can be viewed as a control-flow analysis adapted to an outermost unfolding strategy. While performing this analysis we keep track of the depth in which unfolding occurs and the depth of arguments bound to formal parameters.

The definition is followed by extensive explanations along with an example.

Definition 15 (Constraints approximating deforestation) Let $p \equiv \textbf{letrec}\ d_1; \ldots; d_n\ \textbf{in}\ t_{init}$, let T be the set of all subterms contained in p, and let $A = T \cup \{\bullet\}$, where $\bullet$ is a special symbol.

For program p, we introduce variable $[p]$, as well as five different variables $[t]$, $r[t]$, $d[t]$, $s[t]$ and $a[t]$ for every $t \in T$. The following table shows the type of constraints in which the variables are used.

Variable	Lattice	Constraint Type
$[p], [t]$	2^A	simple
$r[t]$	$\mathbf{2}$	boolean
$d[t]$	$\mathcal{N}$	integer
$s[t]$	$\mathcal{N}$	integer
$a[t]$	$\mathcal{N}$	integer

The set of constraints $\mathcal{C}(p)$ computed for p is the smallest set containing the *initial* constraints, and closed under the

[2] In [44], constraints also of the form $X \supseteq Y$ occur. These have been removed in the present formulation.

[3] Instead of constraint systems, [43, 44] considers systems of equations. This makes no difference w.r.t the minimal model.

transitivity rules, *top-level rules*, and *unfolding rules*. The subset of integer constraints is written $\mathcal{I}(p)$.

Initial constraints:

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

where

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

Transitivity rules:

$$\begin{array}{l}
\text{if } [p] \supseteq t, [t] \supseteq t' \text{ then } [p] \supseteq t' \\
\text{if } [t] \supseteq t', [t'] \supseteq t'' \text{ then } [t] \supseteq t'' \\
\text{if } r[t] \sqsupseteq 1, [t] \supseteq t' \text{ then } r[t'] \sqsupseteq 1; d[t] \geq d[t']; s[t'] \geq s[t]
\end{array}$$

Top-level rules:

$$\begin{array}{ll}
\text{if } [p] \supseteq c\ t_1 \ldots t_n & \text{then } [p] \supseteq t_1, \ldots, t_n \\
\text{if } [p] \supseteq \lambda x.t & \text{then } [p] \supseteq t, [x] \supseteq \bullet \\
\text{if } [p] \supseteq t & \text{then } r[t] \sqsupseteq 1
\end{array}$$

Unfolding rules:

$$\begin{array}{l}
\text{if } r[t] \sqsupseteq 1 \text{ then case } t \text{ of} \\
\quad f: \\
\qquad [t] \supseteq t^f; \\
\quad t_1\ t_2: \\
\qquad r[t_1] \sqsupseteq 1; s[t_1] \geq 1 + s[t]; d[t_1] \geq d[t]; \\
\qquad \text{if } [t_1] \supseteq \lambda x.t' \text{ then } [t] \supseteq t'; [x] \supseteq t_2; a[x] \geq a[t_2]; \\
\qquad \text{if } [t_1] \supseteq \bullet \text{ then } [t] \supseteq \bullet; [p] \supseteq t_2; \\
\quad \mathbf{let}\ x{=}t_1\ \mathbf{in}\ t_2: \\
\qquad [t] \supseteq t_2; [x] \supseteq \bullet; [p] \supseteq t_1; \\
\quad \mathbf{case}\ t_0\ \mathbf{of}\ q_1{\to}t_1; \ldots; q_m{\to}t_m: \\
\qquad r[t_0] \sqsupseteq 1; s[t_0] \geq s[t]; d[t_0] \geq 1 + d[t]; \\
\qquad \text{if } [t_0] \supseteq \bullet \text{ then} \\
\qquad\quad [t] \supseteq t_1, \ldots, t_n; \\
\qquad\quad [y] \supseteq \bullet \text{ (for all } y \text{ in } q_1, \ldots, q_n); \\
\qquad \text{if } [t_0] \supseteq c\ s_1 \ldots s_n \text{ and } q_j \equiv c\ x_1 \ldots x_n \text{ then} \\
\qquad\quad [t] \supseteq t_j; \\
\qquad\quad [x_1] \supseteq s_1; \ldots; [x_n] \supseteq s_n; \\
\qquad\quad a[x_1] \geq a[s_1]; \ldots; a[x_n] \geq a[s_n];
\end{array}$$

□

The meaning of the variables are as follows.

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 operand and selector, respectively.

The *boolean* variable $r[t]$ shows whether transformation of t is forced by a surrounding context.

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 operands 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.

Note that the three subsystems of $\mathcal{I}(p)$ containing variables $a[.]$, $d[.]$, and $s[.]$, respectively, are disjoint,

The effect of the constraints are as follows.

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 $a[x]$, $x \in \mathcal{V}$.

The *transitivity rules* model iterated unfolding.

The *top-level* rules model rules (2-3) of deforestation which push transformation under constructors and abstractions (rule (1) is modelled by the unfolding rules). When going under an abstraction, the abstracted variable obtains the status of a free variable and so receives value $\bullet$.

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 parts of **let**- and **case**-expressions must be raised to the top-level, and 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 16 The program in Example 13 has constraints:

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

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

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

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

Here the integer constraints I include:

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

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

For Example 11 the integer constraints I include

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

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

For Example 12 the integer constraints I include

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

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. □

The following theorem shows that the set of integer constraints in general contains enough information to estimate whether deforestation loops, and that the set can be computed efficiently.

Theorem 17 *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), (2) or (3) holds:*

(1) $\mu I\ a[x] = \infty$ for some variable x;

(2) $\mu I\ d[t] = \infty$ for some subterm t;

(3) $\mu I\ s[t] = \infty$ for some subterm t.

(ii) *I can be computed in polynomial time.*

Proof. See Appendix A. □

Properties (1-2) correspond to the second author's [47, 44] criteria for accumulating parameters and obstructing function calls, respectively, for first-order deforestation. In the higher-order case (3) captures accumulating spines. As we shall see in Section 8, typable programs never give rise to accumulating spines.

7 Generalizing dangerous subterms

Section 6 shows how to guarantee that deforestation terminates on some program: check that conditions (1)-(3) are all false. It remains to show that these conditions can be decided efficiently, and it remains to compute appropriate generalizations in case one of the conditions are true, i.e., when deforestation may fail to terminate.

Given $I = \mathcal{I}(p)$. Any inequality $Y \geq P \in I$, where P is a polynomial built from variables, constants, "+", "$\sqcup$" can be expressed by a set of constraints of the forms $Y \geq c + X$ and $Y \geq c$, where $c \geq 0$ is an integer. Therefore, we may assume that the constraints in I are of these forms.

Next we give a characterization of those X with $\mu I\ X = \infty$.[4] We also make explicit how the set of these variables can be determined efficiently.

Definition 18 (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 $Y \geq c + X \in I$. □

[4] As observed in [44], one may also determine 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 [34].

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 19 *Given p and $I = \mathcal{I}(p)$. For $\tau \in \{a, d, s\}$, let $J_\tau = \{t \mid \mu I\ \tau[t] = \infty\}$. Then*

1. *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 $\tau[t_1], \tau[t_2]$ such that $c \geq 1$ and $\tau[t_1] \geq c + \tau[t_2] \in I$; or*
   - *$\tau[t]$ is reachable in G_I from $\tau[t']$ with $t' \in J_\tau$.*
2. *J_τ can be computed in linear time.*

Proof. See [44], Theorem 2. □

By Proposition 19 we can sharpen the formulations of criteria (2) and (3) in Theorem 17. For criterion (1) we are only able to provide a more concrete form if $a[x']$ receives a finite value for all pattern variables x'.

Corollary 20 *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 a variable x iff some subterm $t \equiv t_1\, t_0$ of p exists where t_0 contains a free variable $z \not\equiv t_0$ and $a[z]$ is in the same strong component of G_I as $a[t_0]$.

(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 $t \equiv t_0\, t_1$ of p exists where $s[t_0]$ is contained in the same strong component of G_I as $s[t]$.

Proof. See Appendix C. □

In view of Corollary 20, 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 21 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 rules:

(1)
- $t \equiv \mathbf{case}\ t_0\ \mathbf{of}\ q_1 \to t_1; \ldots; q_m \to t_m$ and for variable x' of a pattern q_i, $\mu I\ a[x'] = \infty$. Then generalize t_0 at t.
- $t \equiv t_1\, t_0$ and t_0 contains a free variable $z \not\equiv t_0$ and $a[z]$ is in the same strong component of G_I as $a[t_0]$. Then generalize t_0 at t.

(2) $t \equiv \mathbf{case}\ t_0\ \mathbf{of}\ q_1 \to t_1; \ldots; q_m \to t_m$ 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 22 whereas correctness is the contents of Theorem 23.

Theorem 22 Given program p, then at most $|p|$ generalizations are possible.

Proof. See Appendix E. □

Theorem 23 Assume p is a program, $I = \mathcal{I}(p)$.

(1) If no generalization is possible according to rule (1) then $\mu I\ a[x] < \infty$ for all variables x.

(2) If no generalization is possible according to rule (2) then $\mu I\ d[t] < \infty$ for all t.

(3) If no generalization is possible according to rule (3) then $\mu I\ s[t] < \infty$ for all t.

Proof. By Corollary 20. □

Hence, deforestation of some program terminates if no further meaningful generalizations can be applied to it.

8 Relation to higher-order treelessness

Hamilton [22] and later Marlow [34] 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 selectors in **case**-expressions to be variables. The following definition is Hamilton's version.

Definition 24 (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:

$$\begin{array}{l} tt ::= x \mid c\ tt_1 \ldots tt_n \mid \mathbf{case}\ x\ \mathbf{of}\ q_1 \to tt_1; \ldots; q_k \to tt_k \mid \\ \qquad \lambda x.tt \mid t\ x \mid f \mid \mathbf{let}\ v = tt\ \mathbf{in}\ tt' \\ ft ::= x \mid f \mid ft\ ft \\ tp ::= \mathbf{letrec}\ f_1 = tt_1; \ldots; f_n = tt_n\ \mathbf{in}\ \lambda x_1, \ldots, x_m.ft \end{array}$$ □

Note that we do not demand treeless terms to be *linear*. In general, as can be seen by Example 13, 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 in this paper 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 [36, 26]). We write $t : \tau$ to express the fact that t has type τ. Similarly for a program p.

Without loss of generality we may assume for a program $p \equiv \mathbf{letrec}\ d_1; \ldots; d_n\ \mathbf{in}\ t_{init}$ 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.

First, typable programs never give rise to accumulating spines.

Proposition 25 Given a typable program p, let $I = \mathcal{I}(p)$. Then $\mu I\ s[t] < \infty$ for all subterms t of p.

Proof. See Appendix B. □

The following shows that for a typable, higher-order treeless program, our analysis finds that no annotations are required, provided all constructors have non-functional arguments 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 26 Assume $p \equiv \mathbf{letrec}\ d_1, \ldots, d_n\ \mathbf{in}\ t_{init}$ is typable, higher-order treeless, and that all constructors in p have non-functional arguments only. Then conditions (1)-(3) of Theorem 17 are all false.

Proof. See Appendix D. □

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

Example 27 Consider the following program.

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

Unfolding the outer call to I in term $I\,(I\,x)$ leads to the term $\mathbf{case}\ (I\,x)\ \mathbf{of}\ [] \to []; (h : t) \to h : I\ t$ 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:

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

In the first-order case this trick is sufficient to ensure that no generalizations are performed on treeless programs [47]. However, in the higher-order case, the problematic situation may arise after a number of transformation steps as in the program:

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

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. □

An investigation of possible enhancements of our analysis to avoid occasional deficiencies of this type remains for future work.

9 Conclusion

We have given a technique to ensure termination of higher-order deforestation allowing useful transformation steps what were not previously possible. The technique can be efficiently implemented using well-known techniques for constraint systems.

A somewhat different approach to elimination of intermediate data structures in higher-order programs is due to Gill, Launchbury, and Peyton Jones [17, 18, 19] who remove intermediate lists explicitly produced and consumed by means of the primitives `build` and `foldr` within the same function. Similar techniques were independently proposed by Sheard and Fegaras [15, 46], and later by Takano and Meijer [48]. These approaches rely on functions being written in a form that explicitly builds and destroys intermediate data structures, although some functions can be transformed into this form automatically [31]. Gill [17] shows that some programs can be improved by traditional deforestation, but not by the `build-foldr` technique, whereas other programs can be improved by the `build-foldr` technique, but not by traditional deforestation, see also [34]. A more direct comparison remains to be done.

Acknowledgments. 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] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

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

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

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

[22] 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.

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

[24] 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.

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

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

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

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

[29] N.D. Jones. Flow analysis of lazy higher-order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Language*, chapter 5. Ellis Horwood, London, 1987.

[30] 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.

[31] 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.

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

[33] 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.

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

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

[36] 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.

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

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

[39] 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.

[40] 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.

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

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

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

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

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

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

A Proof of correctness of the analysis

This appendix is devoted to the proof of Theorem 17. To prove the first part, we extend the methods of [44] to the higher-order case. First we set up a relation $\Rightarrow$ which gives a more convenient way to deal with the notion of transformation encountering terms.

Definition 28 Let $t^\bullet$ denote the result of replacing all free variables of t by $\bullet$. Define $\Rightarrow$ on programs and closed terms by:

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

In (4) definitions $f = t^f$ must be located among $d_1,\ldots,d_n$ of the program. □

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 simulates 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 $x\ t_1\ldots t_n$.

The following shows that the problem of ensuring that $[\![.]\!]$ of Definition 5 encounters only finitely many different terms (see Remark 10) is equivalent to the problem of ensuring that $p \stackrel{*}{\Rightarrow} s$ for only finitely many different s.

Proposition 29 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 $\stackrel{*}{\Rightarrow}$. □

Next, we introduce a reduction manipulating substitutions and call-by-name contexts explicitly by means of environments and stacks, respectively.

Definition 30 Let E and σ range over environments and stacks, respectively, as defined by the grammar ($n \geq 0$):

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

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. □

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

Definition 31 Define $\to$ on programs, pairs (t, E) and stacks σ by:

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

□

Rules (0a,1a-8) simulate the corresponding rules of Definition 28. 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 32

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

□

By induction on reduction steps we verify:

Proposition 33 Assume $t = u[\sigma]$. Then

1. *$t \Rightarrow t'$ implies $\sigma \xrightarrow{*} \sigma'$ for some σ' with $u[\sigma'] = t'$;*
2. *$\sigma \rightarrow \sigma'$ implies $t = u[\sigma']$ or $t \Rightarrow u[\sigma']$.* □

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 36 and 37 below. To this end we introduce an operator 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 34 On environments and stacks define α by:

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

□

Definition 35 Define the *depth* $|t|$ of a term t as follows.

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

□

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

Proposition 36 Let I, J be the integer constraints in αE, $\alpha\sigma$, respectively.

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

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

Proposition 37 Given p. If $p \xrightarrow{*} \sigma$ then $\alpha\sigma \subseteq \mathcal{C}(p)$. □

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

Proposition 38 Let $u[(t_1, E_1) \ldots (t_k, E_k)] = t$. Suppose that $|t_j| \leq r$ and $|u[E_j(x)]| \leq a$ for all j and $x \in \mathrm{dom}(E_j)$. Then $|t| \leq r + a + k - 1$. □

Propositions 36, 37 and 38 can be combined to obtain:

Proposition 39 Given $p \equiv \mathbf{letrec}\ f_1 = t_1; \ldots; f_n = t_n\ \mathbf{in}\ t_{init}$ 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 37, the set of integer constraints of $\alpha\sigma$ are contained in I. Hence by Proposition 36, $|\sigma| \leq d + s$ 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 38, $|u[\sigma]| \leq r + a + d + s$. □

We are now in a position to prove Theorem 17.

Proof.(Theorem 17). For the first part assume that $[\![p]\!] = \cdots [\![t]\!] \cdots$. By Proposition 29 Then also $p \xRightarrow{*} t^\bullet$. By Proposition 33 $p \xrightarrow{*} \sigma$ for some stack σ with $u[\sigma] = t^\bullet$. By Proposition 39, $|t| = |t^\bullet| = |u[\sigma]| \leq r + a + d + s$, 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 the right hand sides 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 ∞.

For the efficiency part we observe that for our control-flow analysis we can use (an adaptation) of Heintze's algorithm for computing a normalized system of set constraints in [25]. Additionally, we have to maintain the constraints for variables $r[.]$ and generate the integer constraints in I. Note that, theoretically Heintze's algorithm has cubic complexity. In practice, however, we found that it behaves quite well on all example programs.

Finally, by the algorithm in [43, 44] the least model of integer constraints can be computed in linear time. □

B Proof of non-accumulating spines

This appendix is devoted to a proof of Proposition 25.

Proof.(Proposition 25). First, transformation satisfies a *subject reduction property:* whenever $t : \tau$ and $[t] \supseteq t' \in S$ then $t' : \tau$ as well.

Let D denote the set of all types of subterms in p and define the ordering "$\leq$" as the reflexive and transitive closure of relation "$<$" defined by $\tau < \sigma$ iff $\sigma = \tau \rightarrow \tau'$ for some τ'. Now define function R mapping subterms to elements in D by $R[t] = \tau$ iff $t : \tau$. Then we have:

1. $[t] \supseteq t' \in S$ implies $R[t] = R[t']$.
2. Whenever $t_1\, t_2$ is a subterm of p then $R[t_1] > R[t_1\, t_2]$.

From 1 and 2 we deduce 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. □

C Proof of characterization of termination

This appendix is devoted to a proof of Corollary 20.

Proof.(Corollary 20). For statement (1) assume that for every pattern variable x', $a[x']$ receives a finite value. Then every such x can only be contained in strong components having edges corresponding to constraints of the form $a[y] \geq a[y']$.

Now let $\mu I\ a[x] = \infty$ for some variable x. By Proposition 19, 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 $N[t]$. Hence, z must be a free variable of t_2 where $t_2 \not\equiv z$. 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 $z \not\equiv t_2$, $N[t_2]$ has the form $N[t_2] = pl \sqcup (c + a[z])$ for some $c > 0$. But then Q contains an edge corresponding to constraint $a[t_2] \geq c + a[z]$ which, by Proposition 19, 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. □

D Proof of conservativity

This appendix is devoted to a proof of Theorem 26.

Proof.(Theorem 26). Let $C = \mathcal{C}(p)$ and $I = \mathcal{I}(p)$. Since p is typable we know from Proposition 25 that $\mu I\ s[t] < \infty$ for all subterms t. It remains to prove finiteness for variables $a[x]$ and $d[t]$.

As in the proof of Proposition 25, 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

1. $[t] \supseteq t'$ implies $R[t] = R[t']$.
2. 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

1. $[t] \supseteq t'$ implies $R[t] \geq R[t']$.
2. If $t \equiv x$ and $[x] \supseteq t'$ then t' is a variable or $R[x] > R[t']$.

Claim 2:

1. If $a[t_1]$, $a[t_2]$ are in the same strong component of G_I then $R[t_1] = R[t_2]$.
2. If $d[t_1]$, $d[t_2]$ are in the same strong component of G_I then $R[t_1] = R[t_2]$.

The proof of Claim 1 is omitted. We infer Claim 2 from Claim 1 and then show how Theorem 26 is implied by these.

If $a[x] \geq a[t] \in I$ then also $[x] \supseteq t \in C$ and therefore by statements (1) of Claim 1, $R[x] \geq R[t]$. Since also $R[t] = R[z]$ for every variable occurring in t, Claim 2(1) 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 $[t'] \supseteq t_1 \in C$ for a superterm of t_2 with the same rank. Therefore again by Claim 1(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**-expression equals the rank of the **case**-expression itself, Claim 2(2) follows.

Finally, consider Theorem 26. 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 of t_0 only receive values $\bullet$. Therefore, by statements (2) of Claim 1, x must be of non-functional type with $R[x] > R[t]$, contradicting Claim 2(1). Hence we 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. Then especially, t is a **case**-expression with selector t'. Since selectors of **case**-expressions are always variables of non-functional type, we know from statement (2) of the non-functional case in Claim 1 that $R[t] > R[t']$ – in contradiction to assertion (2) in Claim 2. Hence, $\mu I\ d[t] < \infty$ for all t. □

E Proof of termination of generalization

This appendix is devoted to a proof of Theorem 22.

Proof.(Theorem 22). 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

1. at an application $t\ x$;
2. at a **case**-expression **case** x **of** $q_1 \to t_1; \ldots; q_m \to t_m$;
3. at an application $x\ t$.

Here (1) is true since we never generalize arguments that are variables.

For (2), consider a **case**-expression $t \equiv$ **case** x **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 $[z] \supseteq \bullet$. Therefore, 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.

Moreover, no integer constraint is generated whose right hand side contains $d[x]$, Hence, $d[x]$ cannot be contained in a strong component containing at least one edge. Therefore, (2) is true too.

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 (3) is true. □