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 eliminates intermediate data structures from functional programs. To be suitable for inclusion in a compiler, it must terminate on all programs. Several techniques to ensure termination of deforestation on all first-order programs are known, but a technique for higher-order programs was only recently introduced by Hamilton, and elaborated and implemented in the Glasgow Haskell compiler by Marlow. We introduce a new technique for ensuring termination of deforestation on all higher-order programs that allows useful transformation steps prohibited in Hamilton's and Marlowe's techniques.

1 Introduction

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

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 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 therefore prefer a program like the following.

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

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 [17, 54, 55] 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 [7, 8, 10, 11, 13], and later to Hamilton [21, 22, 24, 25]. 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. A technique that annotates fewer parts of the program is due to Sørensen [48], and was later improved by Seidl [45].

These 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 reduce the higher-order case to the well-studied first-order case. Wadler [55] considers programs with *higher-order*

Submitted to the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 15–17, 1997. WORDCOUNT: approx. 4950 excluding appendices and references.

[1] Earlier techniques include [3, 6, 14, 15, 31, 33, 50, 51, 52, 53].

macros. Any such program typable in the Hindley-Milner [27, 36] 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 a higher-order removal transformation [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 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 general higher-order programs is due to Marlow and Wadler [34], who leave open the question of guaranteeing termination. This was addressed by Hamilton [23], 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. These ideas have been elaborated and implemented in the Glasgow Haskell compiler by Marlow [35].

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.

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

The term $\mathit{foldr}\ c\ v\ u$ 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 $\mathit{foldr}\ f\ a\ zs$ in the definition of *foldr* be annotated, 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 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 [48, 45].

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 have been banished into 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:

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

(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. All variables of t in definitions $f = t$ and 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 erroneous terms of form $(c\, t_1 \ldots t_n)\, t$ and $\textbf{case}\ (\lambda x.t)\, \textbf{of}\, q_1{\to}$

$t_1; \ldots; q_n \to t_n$ never arise. The semantics of the language is call-by-need [2].

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 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 $x_1 : \ldots : x_n : []$ we also write $[x_1, \ldots, x_n]$. Substitution of t' for x in t is written $t\{x := t'\}$. □

The **let**-construct is an alternative to annotations. Instead of annotating the dangerous parts of a program and letting 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 leads to less syntactic overhead than working with annotations.

3 The higher-order deforestation algorithm

We shall state the deforestation algorithm by rules for rewriting 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 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 either an observable or decomposes uniquely into a context e and redex r such that $t \equiv e\langle r \rangle$. This provides a way of finding the next subterm to reduce in a term, i.e., the following clauses of $[\![.]\!]$ are mutually exclusive and together exhaustive. The definition of $[\![.]\!]$ is inspired by [23], but the use of **let**-expressions rather than annotations, and of explicit contexts, has reduced the number of rules from 16 to 8.

Definition 5 (Deforestation)

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

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

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

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

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

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

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

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

Given a program **letrec** $d_1; \ldots; d_n$ **in** t we apply transformation $[\![.]\!]$ to t understanding that definitions $f = t^f$ for function calls must be located among $d_1, \ldots, d_n$. □

As is well-known, this algorithm hardly ever terminates. For instance, on the program **letrec** $f = f$ **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. This can be done in a number of ways (cf. [35] for a discussion). What is important is that if the rules of the preceding definition encounter only finitely many *different* terms, then the algorithm with folding terminates. For concreteness we sketch one way of doing this.[2]

Definition 6 (Folding) Let $[\![.]\!]$ take an extra parameter I. In all clauses of the preceding definition but (4), I is passed unchanged to the recursive calls of $[\![.]\!]$. Replace rule (4) of the preceding definition by (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 $\{x_1, \ldots, x_n\} = \text{FV}(e\langle f \rangle)$. Now, given a program **letrec** $d_1; \ldots; d_n$ **in** t we apply $[\![.]\!]$ to t with the empty set for I. This results in a term, but also in a new set of definitions $g = \lambda x_1, \ldots, x_n..[\![e\langle f \rangle]\!]I'$ generated in the process, which are collected into a new program. □

[2] The following algorithm only folds terms with function calls in the redex. For programs that are not Hindley-Milner typable one must allow folding for β-redices as well.

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:

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

Then transformation proceeds as follows.

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

Hence the new program is

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

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

Apart from termination—the topic of this paper—there are two other aspects of correctness for deforestation, namely 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 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 general technique due to Sands [41, 43] can be used to prove this for deforestation [40, 42].

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. In Wadler's [55], Hamilton's [23], and Marlow's [35] approaches this is avoided by considering only linear programs, i.e., programs consisting of functions that do not duplicate their arguments. Some weaker restrictions are adopted in partial evaluation [46, 5] and in work on first-order deforestation [21]. Second, there is the problem of *code duplication*. Unrestrained unfolding may increase the size of a program dramatically. 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 [35].

4 Termination problems in deforestation

Even with folding, deforestation does not always terminate. Below we present the three kinds of problems that can occur. We show that deforestation applied to each of the programs loops indefinitely, and with certain small changes in the programs, deforestation terminates. These changes are called *generalizations*.

Example 8 (The Accumulating Parameter) Consider the following program.

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

The r function returns its argument list reversed. Applied to this program, deforestation loops indefinitely. The problem is that deforestation encounters the progressively larger terms $rr\ zs\ []$, $rr\ zs_1\ [z_1]$, $rr\ zs_2\ [z_2, z_1]$, etc. Since the formal parameter ys of rr is bound to progressively larger terms, we call ys an *accumulating parameter*.

We can solve the problem if we can make sure that deforestation does not distinguish between the terms that are bound to ys. This can be achieved by transforming the program into:

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

Deforestation applied to this program terminates. □

Example 9 (The Obstructing Function Call) Consider the following 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 to this. Deforestation encounters the terms r, $\textbf{case}\ (r\ zs_1)\ \textbf{of}\ \cdots$, $\textbf{case}\ (\textbf{case}\ (r\ zs_2)\ \textbf{of}\ \cdots)\ \textbf{of}\ \cdots$, etc. Because the call to r prevents the surrounding **case**-expressions from being reduced, we call it an *obstructing function call*.

We can solve the problem if we make sure that deforestation does not distinguish between the different terms that occur in the redex position. This can be achieved by transforming the program into:

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

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

Example 10 (The Accumulating Spine) Yet another possibility to prevent deforestation from termination is to recursively create larger and larger 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 some typing disciplines, e.g., simple types. Deforestation applied to the program successively encounters terms f, $f\ x\ x$, $f\ x\ x\ x$, 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' *in* $t\{x := t'\}$. In Example 8, we generalized rr's second argument $(z : ys)$ in the body of the definition for rr. In example 9, we generalized the call $r\ xs$ in the body of the definition of r. Finally, in example 10, the call to f was generalized in the body of the definition of f.

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

5 Constraint Systems

Inspired by work by Jones and co-workers [30, 29, 1, 37], Sørensen [48] approximates the set of terms encountered during first-order deforestation of a program by a set of terms described by a *regular tree grammar*. Inspecting this grammar one can automatically decide whether any of the phenomena in the preceding section occur, and generalize accordingly.

An equivalent means for describing regular tree languages are *set constraints* [26]. These are technically more convenient than regular tree grammars, and have given rise to fast, practical analysis algorithms. Seidl [45] therefore expressed Sørensen's analysis 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 [38] and Palsberg and O'Keefe [39] and, while doing so, to collect a set of *integer constraints*. In this paper we show that this technique can be elegantly extended to the higher-order case. 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

$$X \sqsupseteq f X_1 \dots X_n$$

where $X, X_1, \dots, X_n \in \mathcal{V}$, 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

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

for every constraint $X \sqsupseteq f X_1 \dots X_n \in S$. 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 powerset 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 $a \in A$.[3] 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).

[3] In [45], constraints also of the form $X \supseteq Y$ occur. These have been removed in the present formulation.

Example 11 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 fixpoint iteration may not suffice to compute least models. Seidl [44] presents algorithms that do compute such least models.[4] 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 12 (Constraints approximating deforestation) Let $p \equiv \textbf{letrec}\ d_1, \dots, d_n\ \textbf{in}\ t_{init}$, let T be the set of all subterms contained in p, and let $A = T \cup \{\bullet\}$, where $\bullet$ is a special symbol.

For every $t \in T$ we consider five different variables $[t]$, $r[t]$, $d[t]$, $s[t]$ and $a[t]$. The following table shows the type of constraints in which the variables are used.

Variable	Lattice	Constraint Type
$[t]$	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 *transitivity rules*, *top-level rules*, and *unfolding rules*. The subset of integer constraints is written $\mathcal{I}(p)$.

The *initial constraints* are as follows.

$$\begin{array}{l} [t] \supseteq t \\ r[t_{init}] \sqsupseteq 1 \\ a[t] \geq N[t] \end{array}$$

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

where $N[t]$ is defined by:

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

The *transitivity rules* are as follows.

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

The *top-level rules* are as follows.

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

The *unfolding rules* are as follows.

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

□

The meaning of the variables are as follows. The variable $[t]$ represents a superset of the terms encountered when transforming t; $\bullet$ denotes a term whose result will be unknown at transformation time, e.g., a free variable, or a free variable applied to some term. The variable $r[t]$ records whether t is unfolded during transformation. The expression $s[t] + d[t]$ is an upper bound on the depth of reduction contexts in which t occurs during transformation, where $s[t]$ counts the nesting inside applications and $d[t]$ counts the nesting inside selectors of **case**-expressions. While variables $s[t]$ have no analog in the first-order case, they are used here to preclude accumulating spines. Finally, $a[t]$ is an upper bound on the depth of t within any term bound to a formal parameter.

The effect of the constraints are as follows. The initial constraints $[t] \supseteq t$ model reflexivity of iterated unfolding, $r[t_{init}] \sqsupseteq 1$ 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 the fact that on the top-level, deforestation is pushed down through constructors and λ-abstraction. When going down under an abstraction, the abstracted variable obtains the status of a free variable and hence receives value $\bullet$.

The unfolding rules approximate the steps of deforestation. First, the rules model indiviual steps of the deforstation 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 deforestion is propagated, i.e., the $r[.]$-information is propagated to the function part of applications and to the selector of **case**-expressions. Third, evaluation of certain subterms of **let**-expressions and case-expressions must be raised to the top-level. Also, the information involving $\bullet$ must be propagated. The main point, however, is to record information about the depth of contexts and arguments. Thus, integer constraints on $a[.]$-variables are generated whenever variables become bound by reductions of β-redexes and **case**-expressions. When passing control to the function part of an application or the selector of a case-expression, the increase in the nesting of the context is recorded by a constraint on the $s[.]$- and $d[.]$-variables, respectively.

Example 13 The program in Example 10 has constraints:

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

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

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

Here

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

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

The program in Example 8 has constraints including

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

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

The program in Example 9 has constraints including

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

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 14 Let $p \equiv \textbf{letrec}\ d_1, \ldots, d_n\ \textbf{in}\ t_{init}$ *and* $I = \mathcal{I}(p)$ *with least model* μI.

(i) *If deforestation applied to 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) and (2) correspond to Sørensen's [48, 45] criteria for accumulating parameters and obstructing function calls, respectively, for first-order deforestation. In the higher-order case (3) is sufficient to capture accumulating spines. However, if p is monomorphically typable,[5] then all values of $s[.]$-variables are finite! In this case accumulating parameters and obstructing function calls are the only reasons for non-termination of deforestation. W.l.o.g. let us assume henceforth that t_{init} does not occur as a subterm of any function definition. Then we have:

Proposition 15 Given a monomorphically typable program p, let $I = \mathcal{I}(p)$ *with least solution* μI. *Then* $\mu I\ s[t] < \infty$ *for all subterms* t *of* p.

Proof. See Appendix B. □

[5] If p is Hindley-Milner typable program, then by duplication of function definitions one obtains a monomorphically typable program.

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.

Let p be a program and $I = \mathcal{I}(p)$. 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.

Definition 16 (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 $Y \geq c + X \in I$. □

Since the three subsystems of I containing variables $a[.]$, $d[.]$, and $s[.]$ are disjoint, Theorem 14 gives the following characterization of those X with $\mu I\ X = \infty$.[6]

Proposition 17 Given p and $I = \mathcal{I}(p)$ *with least model* μI. *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* $\tau[t_1], \tau[t_2]$ *with* $c \geq 1$ *and* $\tau[t_1] \geq c + b[t_2] \in I$; *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 10. □

By Proposition 17 we can sharpen the formulations of criteria (2) and (3) in Theorem 14. 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 18 Given p and $I = \mathcal{I}(p)$ *with least model* μI.

(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* $t \equiv t_1\ t_2$ *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* $d[t_0]$.

[6] As observed in [45], 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 [35].

(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. See Appendix C. □

In view of this we now propose the following strategy for computing generalizations. Starting with program p, we repeatedly generalize subterms of p. In order to determine candidates for generalization, we compute the set $I = \mathcal{I}(p)$ of integer constraints according to our analysis. If μI is finite for all variables $a[t], d[t]$ and $s[t]$ no further generalizations are necessary. Otherwise, we generalize according to the following three rules.

(1)
- $t \equiv t_1\, t_2$ and t_2 contains a free variable $x \notin t_2$ and $a[x]$ is in the same strong component of G_I as $a[t_2]$. Then generalize t_2 in t.
- $t \equiv$ **case** t_0 **of** $q_1 \to t_1; \ldots; q_m \to t_m$ and for a variable x of a pattern q_i, $\mu I\ a[x] = \infty$. Then generalize t_0 in 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 in t.

(3) $t \equiv t_1\, t_2$ and $s[t_1]$ is contained in the same strong component of G_I as $s[t]$. Then generalize t_1 in t.

Our strategy is non-deterministic. We prove that *every* admissible sequence of choices results in a correctly annotated program. If p is monomorphically typable, no generalizations according to (3) will ever be performed.

The only simple constraints generated for a **let**-bound variable x is $[x] \supseteq \bullet$. Therefore, $s[x]$ and $d[x]$ cannot occur in right-hand sides of integer constraints. Also, if the selector of a **case**-expression t is a **let**-bound variable then $[z] \supseteq \bullet$ are the only simple constraints generated for pattern variables z of t, and therefore always $\mu I\ a[z] = 0$. From this we can conclude that no subterm t is ever generalized which already equals a single **let**-bound variable. Hence, every sequence of generalizations eventually terminates. Furthermore, we obtain from Corollary 18:

Theorem 19 Assume p *is a program,* $I = \mathcal{I}(p)$ *is the set of integer constraints generated for* p *and* μI *is the least solution of* I.

(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. □

From Theorem 14 we therefore derive that deforestation definitely terminates when no further meaningful generalizations can be applied to p.

8 Relation to higher-order treelessness

Hamilton [23] and later Marlow [35] 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 20 (Treeless programs) We 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:

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

□

Note that we do not demand treeless terms to be *linear*. In general, as can be seen by Example 10, 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 here programs that are monomorphically typable.

Without loss of generality we may assume for a program $p \equiv$ **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.

The following shows that given a monomorphically 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 21 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)-(3) of Theorem 14 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 22 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 term $\textbf{case}\ (I\ x)\ \textbf{of}\ [] \to []; (h:t) \to h : I\ t$ 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 [48]. 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. □

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 [18, 19, 20] 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 [16, 47], and later by Takano and Meijer [49]. 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 [32]. Gill [18] 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 [35]. 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] N. Andersen. Approximating term rewrite systems. Technical Report 86/16, Department of Computer Science, University of Copenhagen, 1986.

[2] 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.

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

[4] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, Amsterdam, 1988.

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

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

[25] 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.

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

[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] N.D. Jones and S.S. Muchnick. Flow analysis of LISP-like structures. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 244–256. ACM Press, 1979.

[31] 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.

[32] 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.

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

[34] 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.

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

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

[37] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In Bjørner et al. [4].

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

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

[40] 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.

[41] 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.

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

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

[44] 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.

[45] 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.

[46] P. Sestoft. Automatic call unfolding in a partial evaluator. In Bjørner et al. [4], pages 485–506.

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

[48] 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.

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

[50] 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.

[51] 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.

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

[53] 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.

[54] 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.

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

A Proof of correctness of the analysis

Proof of Theorem 14. To prove correctness of our analysis, we extend the methods of [45] to the higher-order case. First, we replace all free variables of terms t to be transformed by the special symbol "$\bullet$". Then we put up a reduction relation "$\Rightarrow$" on such closed terms such that if transformation of t leads to transformation of t' then also $t \stackrel{*}{\Rightarrow} t'$.

Definition 23

$$
\begin{array}{lcl}
c\,t_1 \ldots t_n & \Rightarrow & t_i \\
\lambda x.t & \Rightarrow & t\{x := \bullet\} \\
e\langle \bullet\, t\rangle & \Rightarrow & e\langle \bullet\rangle \\
e\langle \bullet\, t\rangle & \Rightarrow & t \\
e\langle f\rangle & \Rightarrow & e\langle t^f\rangle \quad (f = t^f) \\
e\langle (\lambda x.t)\, t'\rangle & \Rightarrow & e\langle t\{x := t'\}\rangle \\
e\langle \textbf{let } x{=}t \textbf{ in } t'\rangle & \Rightarrow & e\langle t\{x := \bullet\}\rangle \\
e\langle \textbf{let } x{=}t \textbf{ in } t'\rangle & \Rightarrow & t \\
e\langle \textbf{case } (c\,t_1 \ldots t_n) \textbf{ of } q_1{\to}s_1; \ldots; q_m{\to}s_m\rangle & \Rightarrow & \\
e\langle s_j\{x_1 := t_1, \ldots, x_n := t_n\}\rangle \quad (q_j \equiv c\; x_1 \;\ldots x_n) & & \\
e\langle \textbf{case } \bullet \textbf{ of } q_1{\to}s_1; \ldots; q_m{\to}s_m\rangle & \Rightarrow & \\
e\langle s_i\{x_1 := \bullet, \ldots, x_n := \bullet\}\rangle \quad (q_i = c\; x_1 \ldots x_n) \quad \textit{for all } i & & \\
\end{array}
$$

□

Note that expressions of the form $\bullet\, t_1 \ldots t_n$ are successively rewritten to $\bullet$ while their argument terms t_i are raised to the top-level.

Instead of abstracting this reduction system we prefer to abstract the corresponding system where substitutions are made explicit. This allows us, for instance, to keep track of instances of subterms. Therefore instead of terms we consider *stacks* of pairs consisting of terms and environments. Environments E are used for making substitutions explicit whereas stacks σ are introduced to model the outermost unfolding strategy. The stack stores suspended abstractions and **case**-expressions.

Definition 24 Let E and σ range over environments and stacks, respectively, as defined by the grammar:

$$
\begin{array}{rcl}
E & ::= & \emptyset \mid \{x := (t, E)\} + E' \\
\sigma & ::= & (t, E)\; \sigma' \\
\sigma' & ::= & \epsilon \mid (t\; t', E)\; \sigma' \\
 & \mid & (\textbf{case } t_0 \textbf{ of } q_1{\to}t_1; \ldots; q_m{\to}t_m, E)\; \sigma'
\end{array}
$$

As usual, we abbreviate

$$\{x_1 := (t_1, E_1)\} + \ldots + \{x_n := (t_n, E_n)\} + \emptyset$$

by $\{x_1 := (t_1, E_1), \ldots, x_n := (t_1, E_n)\}$.

The function $u[.]$ maps a pair consisting of a term and an environment (or a stack of such pairs) to the term that the pair (resp. the stack) denotes.

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

□

The following proposition, which relates the depth of terms with depth of stacks and environments, can be proved by induction on the depth of terms.

Proposition 25 Let a_0 denote the maximal depth of a term in p, and assume we are given term t and stack $\sigma = (t_1, E_1) \ldots (t_k, E_k)$ containing only subterms from p and $\bullet$ such that $u[\sigma] = t$. If $\mathrm{depth}(u[E_j\; x]) \leq a$ for all j and $x \in \mathrm{dom}(E_j)$ then

$$\mathrm{depth}(t) \leq k + a_0 + a$$ □

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

Definition 26 Define $\to$ by:

$$
\begin{array}{lcl}
(x, E)\; \sigma & \to & (E\; x)\; \sigma \\
(c\,t_1 \ldots t_n, E) & \to & (t_i, E) \quad \text{for all } i \\
(\lambda x.t, E) & \to & (t, \{x := (\bullet, \emptyset)\} + E) \\
(f, E)\; \sigma & \to & (t^f, \emptyset)\; \sigma \\
(t\; t', E)\; \sigma & \to & (t, E)\; (t\; t', E)\; \sigma \\
(\bullet, \emptyset)\; (t\; t', E)\; \sigma & \to & (\bullet, \emptyset)\; \sigma \\
(\bullet, \emptyset)\; (t\; t', E)\; \sigma & \to & (t', E) \\
(\lambda x.t, E)\; (t_1\; t_2, E_2)\; \sigma & \to & (t, E + \{x := (t_2, E_2)\})\; \sigma \\
(\textbf{let } x{=}t \textbf{ in } t', E)\; \sigma & \to & (t', E + \{x := (\bullet, \emptyset)\})\; \sigma \\
(\textbf{let } x{=}t \textbf{ in } t', E)\; \sigma & \to & (t, E) \\
(\textbf{case } t_0 \textbf{ of } q_1{\to}t_1; \ldots; q_m{\to}t_m, E)\; \sigma & & \\
\quad \to (t_0, E)\; (\textbf{case } t_0 \textbf{ of } q_1{\to}t_1; \ldots; q_m{\to}t_m, E)\; \sigma & & \\
(c\, s_1 \ldots s_n, E)\; (\textbf{case } t_0 \textbf{ of } q_1{\to}t_1; \ldots; q_m{\to}t_m, E_1)\; \sigma & & \\
\quad \to (t_j, E')\; \sigma \text{ whenever } q_j \equiv c\; x_1, \ldots, x_n & & \\
\quad \text{and } E' = E_1 + \{x_1 := (s_1, E), \ldots, x_n := (s_n, E)\} & & \\
(\bullet, \emptyset)\; (\textbf{case } t_0 \textbf{ of } q_1{\to}t_1; \ldots; q_m{\to}t_m, E_1)\; \sigma & & \\
\quad \to (t_i, E')\; \sigma \quad \text{for all } i \text{ whenever } q_i \equiv c\; x_1, \ldots, x_n & & \\
\quad \text{and } E' = E_1 + \{x_1 := (\bullet, \emptyset), \ldots, x_n := (\bullet, \emptyset)\} & &
\end{array}
$$

□

By case distinction on reduction steps we verify:

Proposition 27 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* $t \Rightarrow u[\sigma']$. □

From Proposition 27 we conclude that it is enough to abstract relation "$\rightarrow$" on stacks. For stack σ, let $\alpha\sigma$ denote the following set of simple constraints.

- If $\sigma = \epsilon$ then $\alpha\sigma = \emptyset$;
- If $\sigma = (t, E)$ then $\alpha\sigma = \{[t_{init}] \supseteq t\} \cup \alpha E$;
- If $\sigma = (t, E)(t_1\ t_2, E')\ \sigma'$ then $\alpha\sigma = \{[t_1] \supseteq t\} \cup \alpha E \cup \alpha((t_1\ t_2, E')\ \sigma')$;
- If $\sigma = (t, E)$ (**case** t_0 **of** $q_1 \rightarrow t_1; \ldots; q_m \rightarrow t_m, E')\ \sigma'$ then $\alpha\sigma = \{[t_0] \supseteq t\} \cup \alpha E \cup \alpha((\textbf{case}\ t_0\ \textbf{of}\ q_1 \rightarrow t_1; \ldots; q_m \rightarrow t_m, E')\ \sigma')$

where

$$\alpha E = \begin{cases} \emptyset & \text{if } E = \emptyset \\ \{[x] \supseteq t\} \cup \alpha E_1 \cup \alpha E_2 & \text{if } E = \{x{:=}(t, E_1)\} + E_2 \end{cases}$$

By induction on the length of reduction paths we prove:

Proposition 28 Let $I = \mathcal{I}(p)$ *with least model* μI. *Assume* $(t_{init}, \emptyset) \xrightarrow{*} \sigma$ *where* $\sigma = (t, E)\ \sigma'$. *Then*

1. $\alpha\sigma \subseteq S$;
2. $r[t] \sqsupseteq 1$ *is in* S;
3. $(\mu I\ d[t]) + (\mu I\ s[t]) \geq |\sigma'|$;
4. $\mu I\ a[x] \geq \mathrm{depth}(u[x, E])$ *for every* $x \in \mathrm{dom}(E)$;
5. $\mu I\ a[t] \geq \mathrm{depth}(u[t, E])$.

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

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

We conclude:

Theorem 29 Consider program $p \equiv$ **letrec** *defs* **in** t_{init} *and* $I = \mathcal{I}(p)$ *with least model* μI.

Safety: Assume during deforestation of p, t *is encountered. Then* $(t_{init}, \emptyset) \xrightarrow{*} \sigma$ *for some* $\sigma = (t', E')\ \sigma'$ *with* $u[\sigma] = t\{x_1 := \bullet, \ldots, x_k := \bullet\}$ $(\{x_1, \ldots, x_k\} = FV(t))$ *such that:*

1. $(\mu I\ d[t']) + (\mu I\ s[t']) \geq |\sigma'|$;
2. $\mu I\ a[x] \geq \mathrm{depth}(u[E\ x])$ *for all* E *occurring in* σ *and all* $x \in \mathrm{dom}(E)$.

Efficiency: S *can be computed in polynomial time.*

Proof. The safety part follows from Proposition 28. The fast algorithm to perform the control-flow analysis essentially consists (of an adaptation) of Heintze's algorithm for computing a normalized system of set constraints in [26] where additionally the constraints for variables $r[.]$ are considered and the integer constraints in I are generated. Note that, theoretically Heintze's algorithm has cubic complexity. In practice, however, we found that it behaves quite well on all example programs. □

Note that μI indeed contains all information necessary to determine whether or not $[\![.]\!]$ stops. Additionally applying Proposition 25 we obtain Theorem 14. This concludes the proof of Theorem 14.

B Proof of non-accumulating spines

Proof of Proposition 15. First note that control-flow analysis is conservative over types, i.e., 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. If $t \not\equiv t_{init}$ then $[t] \supseteq t' \in S$ implies $R[t] = R[t']$.
2. Whenever $t_1\ t_2$ is a subterm of p then $R[t_1] > R[t_1\ t_2]$.

The special treatment of t_{init} is due to the possibilities of "raising" subterms to the top-level - which clearly does not preserve types. 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.

This concludes the proof of Proposition 15.

C Proof of characterization of termination

Proof of Corollary 18. 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 17, some strong component Q exists which contains an edge corresponding to a constraint $a[t] \geq c + a[z]$ with $c > 0$. Since z' cannot be a pattern variable, this constraint must have been generated for an application $t_1\ t$ where $[t_1] \supseteq \lambda z'.t'$, i.e., $N[t] = pl \sqcup (c + a[z])$ for some polynomial pl. Especially, $a[z]$ is a free variable of $N[t]$. Hence, z must be a free variable of t where $t \not\equiv z$. 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 17, 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 inceased precisely when going from an application to its operator.

This finishes the proof of Corollary 18.

D Proof of conservativity

Proof of Theorem 21. Let S and I denote the sets of constraints resp. integer constraints generated by our analysis applied to p. Furthermore, let μI denote the least solution of I. Since p is monomorphically typed we know from Proposition 15 that $\mu I\ s[t] < \infty$ for all subterms t. Therefore, it remains to prove finiteness for variables $a[x]$ and $d[t]$.

As in the proof of Proposition 15, 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 our extra assumption, t_0 is contained in D and is the maximal element.

Given D, function R is 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 $R[t] = d$ for every subterm t occurring in the right-hand side of f. Now assume $t' \neq \bullet$. Then the following holds:

1. If $[t] \supseteq t'$ then $R[t] \geq R[t']$.
2. If variable x is of functional type, then $a[x] \geq a[t'] \in I$ implies that either t' is a variable as well or a subterm of t_0.
3. If variable x is *not* of functional type, then $[x] \supseteq t'$ implies that either t' is a variable or $R[x] > R[t']$.

From statement 3 we deduce that for all t, $\mu I\ d[t]$ is bounded above by the height of D plus 1. For the $a[.]$-constraints statements 2 and 3 imply that every strong component has only edges corresponding to constraints of the form $a[x] \geq a[y]$. It follows that $\mu I\ a[x] < \infty$ for all variables x.

This concludes the proof of Theorem 21.