A New Means of Ensuring Termination of Deforestation

Student Project

Morten Heine Sørensen

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

August 4, 1993

Abstract

The *deforestation* algorithm transforms functional programs which use intermediate data structures into semantically equivalent programs which do not use intermediate data structures. However, the deforestation algorithm is only guaranteed to terminate for *treeless* programs. The *generalizing deforestation* algorithm does the same job as the standard deforestation algorithm except that it leaves subterms which are annotated with $\ominus$ untransformed. The problem remains to give the program *safe* annotations, *i.e.* annotations ensuring that application of the generalizing deforestation algorithm to the annotated program terminates.

We develop a method of finding safe annotations automatically. Given a program, the idea is to calculate a grammar such that (at least) every term that the deforestation algorithm encounters when transforming the program is derivable from the grammar. Whenever the deforestation algorithm loops infinitely, it encounters infinitely many different terms, and whenever the deforestation algorithm encounters infinitely many different terms, infinitely many different terms are derivable from the grammar.

The problem of deciding whether the deforestation algorithm loops infinitely for arbitrary programs is undecidable, but the problem of deciding whether infinitely many different terms are derivable from an arbitrary grammar is decidable. From the grammar suitable annotations can therefore be calculated automatically.

Our method seems to perform well compared to previous methods. Specifically, we show that if the program is treeless our method will find that no annotations are required, and we conjecture that our method can be extended to never find worse annotations than Chin's method. By way of an example we describe a class of non-treeless programs which do not require annotations by our method. This shows that our method strictly improves Wadler's method, which consists of restricting application of the deforestation algorithm to treeless programs. The example program requires annotations by Chin's method. Taking the conjecture above for granted, this shows that our method strictly improves Chin's method.

Preface

This paper constitutes a student project ("skriftligt arbejde") at DIKU, the Department of Computer Science at the University of Copenhagen. It reports work done between August 1992 and March 1993. My advisor was Neil D. Jones, DIKU.

Chapters 1,2,3 and 5 and Sections 10.1 to 10.5 contain expositions of known material. The problems and solutions of chapter 4 are also generally known although I believe that I have contributed slightly to the insights. The remaining chapters contain original material. This material draws on previous work.

The paper is essentially self-contained.

Acknowledgements

I would like to thank Nils Andersen for discussions on Chin's work. I would like to thank Wei-Ngan Chin for patiently answering my questions concerning his work; any misperceptions left in the exposition in Section 10.3 and 10.4 are of course entirely my responsibility. I would also like to thank G.W. Hamilton for answering questions concerning his work, although I have not yet compared his method to the one presented in this paper. I should also thank David Sands for encouraging me to investigate something similar to the idea in Section 9.1 when I did not think it would work. Finally I would like to thank Neil for asking me interesting questions that I think could only come from an expert in the field. And last of all I would like to thank my friend and lover Gurli who remained a source of energy and joy when I was wandering in the unknown.

Contents

Chapter 1

Introduction

Modern functional programming languages such as Miranda[1] [Tur90] lend themselves to a certain elegant style of programming. Among the features supporting this style of programming, *higher-order functions* and *lazy evaluation* are perhaps the most important; [Hug90a] gives illuminating examples.

While these features make it easy to read and write programs, they are also sources of inefficiency. Consequently, researchers are struggling to find techniques to transform an (elegant) inefficient program into a semantically equivalent (inelegant) efficient program.

Techniques for translating a higher-order program into a semantically equivalent first-order program constitute a current research topic [Nel91]. Eliminating laziness amounts to deciding whether it is safe to evaluate an argument of a function before unfolding the call. This is the purpose of one of the most intensively studied program analyses, strictness analysis [Hug88], [Hug90b].

There are other principles of programming which are similar to those mentioned above, with respect to both elegance and inefficiency; this paper is concerned with one particular such principle, the use of *intermediate data structures.*

EXAMPLE **1** Consider the Miranda program that computes the sum of the

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

squares of the first n natural numbers: (from [Wad88])

$$
\begin{array}{lcll}
ss\,n & = & sum\,(map\,square\,(upto\,1\,n)) & \\
 & & & \\
sum\,Nil & = & 0 & \\
sum\,(Cons\,x\,xs) & = & x + sum\,xs & \\
 & & & \\
map\,f\,Nil & = & Nil & \\
map\,f\,(Cons\,x\,xs) & = & Cons\,(f\,x)\,(map\,f\,xs) & \\
 & & & \\
square\,m & = & m * m & \\
 & & & \\
upto\,m\,n & = & Nil & \text{if } m > n \\
 & = & Cons\,m\,(upto\,(m+1)\,n) & \text{otherwise}
\end{array}
$$

The expression $ss\,n$ is evaluated as follows.

$$
\begin{array}{ll}
ss\,n & \Rightarrow \\
sum\,(map\,square\,(upto\,1\,n)) & \Rightarrow \\
sum\,(map\,square\,(Cons\,1\,(upto\,2\,n))) & \Rightarrow \\
sum\,(Cons\,(square\,1)\,(map\,square\,(upto\,2\,n)) & \Rightarrow \\
(square\,1)\;+\;(sum\,(map\,square\,(upto\,2\,n)) & \ldots
\end{array}
$$

The inner subexpression $upto\,1\,n$ builds a term with a $Cons$ constructor. This term is processed by the surrounding expression, which destroys the constructor term and builds a new one, in this case with the same kind of constructor. This allocation and deallocation of storage at run-time is expensive. Sacrificing clarity for efficiency, we would prefer a program like the following.

$$
\begin{array}{lcll}
h\,0\,1\,n & & & \\
\text{where} & & & \\
h\,a\,m\,n & = & a, & \text{if } m > n \\
 & = & h\,(a + square\,m)\,(m+1)\,n, & \text{otherwise}
\end{array}
$$

The inefficiency has been removed by eliminating the intermediate lists. □

A technique to transform programs of the former kind into the latter, *deforestation,* has been invented by Philip Wadler [Wad88], [Fer88]. In the

example above the intermediate data structure is a list, and indeed Wadler's deforestation algorithm which handles general data structures is a successor of his earlier work on a method of eliminating intermediate *lists* [Wad84], [Wad85]. Both of these techniques were inspired by the early pioneering work by Turchin [Tur80], [Tur82] on the supercompiler, a program transformation technique which does both partial evaluation[2] and deforestation.

We note here that the emphasis is on *fully* automatic methods. General *fold/unfold transformations* similar to deforestation have been known for a long time [Bur77], [Dar81], [Fea82]; these need, however, information from the user. If the technique is to be applied in an optimizing compiler it is necessary that the method be fully automatic. And to apply the technique in an optimizing compiler is the highest of our hopes; this way the programmer can write elegant code, and have it efficiently implemented right away by the compiler.

In this respect, the original deforestation algorithm had one problem: it didn't terminate on all programs. Later, fully automatic methods for achieving this have been described by Chin [Chi90], [Chi92b], [Chi93a] and Hamilton and Jones [Ham90]. These methods work essentially by telling the deforestation algorithm to leave certain subterms untransformed. Obviously, as few subterms as possible should be left untransformed. The main purpose of this paper is to describe a new semantic analysis whose result can be used to ensure termination of deforestation in such a way that fewer subterms will be left untransformed, compared to (some of) the methods above.

The remainder of the paper is organized as follows. Chapter 2 describes a simple pattern-matching first-order language. Chapter 3 describes the deforestation algorithm, its underlying intuition, and the correctness issues pertaining to the transformation. Chapter 4 describes the different problems of ensuring that the deforestation algorithm terminates. Chapter 5 describes the problem of ensuring that the output program of the deforestation algorithm is at least as efficient as the input program. Chapter 7 describes our main contribution, a new semantic analysis whose result can be used to ensure termination. Chapter 6 and 8 show that the analysis is safe. Chapter 9 describes a number of extensions which improve the accuracy of the analy-

[2] We occasionally assume that the reader is familiar with partial evaluation. The paper can, however, be read without such knowledge with almost the same profit. An introduction to partial evaluation can be found in [Jon92].

sis. Chapter 10 describes previous methods of ensuring termination of the deforestation algorithm and relates them to our method. Chapter 11 finally concludes and indicates ramifications of the ideas in this paper.

Chapter 2

A simple first-order pattern-matching language

This chapter describes the language that all our algorithms will be concerned with.

The first section introduces the syntax of our language. The next reviews some notational conventions that we shall employ throughout the paper. The last section describes the operational semantics of our language by a simple rewrite interpreter.

2.1 Syntax

The syntax of our language is defined by the same grammar as in [Fer88].

DEFINITION **1** (Language.) We assume mutually disjoint infinite sets of variable names, constructor names, f-function names and g-function names, ranged over by the generic variables v, c, f and g, respectively. We use h to range over functions which are either f- or g-functions. The generic variables t, p, d, s range over terms, patterns, definitions and programs, respectively. The set of all terms, patterns, definitions and programs are denoted T, P, D

and S, respectively.

$$
\begin{array}{rcll}
t & ::= & v & \text{(variable)} \\
 & | & c\, t_1 \ldots t_n & \text{(constructor)} \\
 & | & f\, t_1 \ldots t_n & \text{(function)} \\
 & | & g\, t_0 \ldots t_n & \text{(function with pattern matching)} \\
 & & & \\
p & ::= & c\, v_1 \ldots v_n & \text{(simple pattern)} \\
 & & & \\
d & ::= & f\, v_1 \ldots v_n = t & (f\text{-function definition}) \\
 & | & g\, p_1\, v_1 \ldots v_n = t_1 & \\
 & & \vdots & (g\text{-function definition with pattern}) \\
 & & g\, p_m\, v_1 \ldots v_n = t_m &
\end{array}
$$

As is customary, we require that patterns be *linear*, *i.e.* that no variable occur more than once. We also require that all variables of a right hand side of a definition be present in the corresponding left hand side. To ensure uniqueness of reduction, we require from a program that each function have at most one definition and, in the case of a g-definition, that no two patterns p_i and p_j contain the same constructor. □

We close this section with some reflections on the significance of the restrictions of our language.

First, the language is first-order. This is not a serious restriction inasmuch as methods exist for transforming a large class of higher-order programs into semantically equivalent first-order programs, *c.f. e.g.* [Nel91]. The applicability of this method depends on how well the (first-order) deforestation algorithm handles the first-order translation of higher-order constructs. Alternatively, Chin [Chi92a], Wadler and Marlow [Mar92] and Hamilton [Ham93] show how one can extend the deforestation algorithm to deal with higher-order features. With such a deforestation algorithm, any analysis which attempts to determine such properties as *e.g.* whether the algorithm terminates, must also take the higher-order language features into account.

Second, the language admits only *simple* patterns, *i.e.* patterns which are exactly one constructor deep. Also, each function definition must not contain more that one pattern matching argument. Again, methods exist for transforming an arbitrary (first-order) program to this form [Aug85],

[Wad87b], and again we should ask ourselves whether the deforestation algorithm handles the transformed programs sensibly, or whether it might be more profitable to extend the deforestation technique to the larger language, at the expense of our analysis becoming more complicated.

We prefer to state our analysis in terms of the simple first-order language for simplicity. We have not investigated whether any significant new problems arise when the analysis is to be conducted in a language with higher-order functions or general patterns.

2.2 Some notational conventions

Our transformation algorithms will be stated as functions[1] of two arguments: a program and a term. The algorithms will be interpreter-like, their effect being the evaluation of the argument term in the argument program. The algorithms will generally contain recursive calls, but always with the original program argument. Therefore the program argument is usually not written explicitly; the algorithm is to be understood in the context of some program. Occasionally we choose to make the context explicit by subscripting the algorithm with the context program.

The following abbreviations are rather tedious to get acquainted with, but they save a lot of explanations later on.

NOTATION **1** If a program s contains a function definition

$$f\ v_1 \ldots v_n = t$$

then in the context of s, t^f denotes t and $v_1^f \ldots v_n^f$ denote $v_1 \ldots v_n$. Similarly, if a program, s, contains a function definition

$$\begin{aligned} g\ p_1\ v_1^1 \ldots v_n^1 &= t_1 \\ &\vdots \\ g\ p_m\ v_1^m \ldots v_n^m &= t_m \end{aligned}$$

[1] Note that now two notions of function are in play: functions in the *subject language*, *i.e.* the language of Definition 1, and functions in the *object language* which is an informal metalanguage used for stating the transformation algorithms. To avoid confusion we henceforth call the latter *algorithms*. We shall be imprecise as to whether algorithms are mathematical functions or functions in a computer language with an operational semantics; accordingly we shall use terms such as "defined" and "guaranteed to terminate" interchangably. We use $\Rightarrow$ to denote evaluation or denotion in the metalanguage.

where $p_i = c_i\ v^i_{n+1} \ldots v^i_{n+k_i}$, then in the context of s, t^{g,c_i} denotes t_i, and $v^{g,c_i}_1 \ldots v^{g,c_i}_n$ denote $v^i_1 \ldots v^i_n$. Finally, $v^{g,c_i}_{n+1} \ldots u^{g,c_i}_{n+k_i}$ denote $v^i_{n+1} \ldots u^i_{n+k_i}$, and $p^g_1 \ldots p^g_m$ denote $p_1 \ldots p_m$. □

EXAMPLE **2** In the context of the program:

$$\begin{array}{lcl} a\ Nil\ ys & = & ys \\ a\ (Cons\ x\ xs)\ zs & = & Cons\ x\ (a\ xs\ zs) \end{array}$$

$t^{a,Nil}$ denotes ys and $t^{a,Cons}$ denotes $Cons\ x\ (a\ xs\ zs)$. Further, $v^{a,Nil}_1$ denotes ys and $v^{a,Cons}_1$ denotes zs. There is no $v^{a,Nil}_2$, while $v^{a,Cons}_2, v^{a,Cons}_3$ denote x, xs. Finally, p^a_1, p^a_2 denote $Nil, (Cons\ x\ xs)$. □

2.3 Semantics

The semantics for reduction of a ground, *i.e.* variable-free, term is the usual reduction to normal form, with top down sequential pattern matching, stated in detail below.

DEFINITION **2** (Ground term, context, redex, observable.) Let $\underline{t}$, $\underline{e}$, $\underline{r}$, $\underline{o}$ be generic variables ranging over *ground terms*, *ground (lazy evaluation) contexts*, *ground redexes* and *ground observables*, respectively, as defined by the grammar:

$$\begin{array}{lcl} \underline{t} & ::= & c\ \underline{t}_1 \ldots \underline{t}_n \mid f\ \underline{t}_1 \ldots \underline{t}_n \mid g\ \underline{t}_0 \ldots \underline{t}_n \\ \underline{e} & ::= & [] \mid g\ \underline{e}\ \underline{t}_1 \ldots \underline{t}_n \\ \underline{r} & ::= & f\ \underline{t}_1 \ldots \underline{t}_n \mid g\ (c\ \underline{t}_{n+1} \ldots \underline{t}_{n+m})\ \underline{t}_1 \ldots \underline{t}_n \\ \underline{o} & ::= & c\ \underline{t}_1 \ldots \underline{t}_n \end{array}$$

The set of all ground terms, contexts, redexes and observables are denoted $\underline{T}$, $\underline{E}$, $\underline{R}$ and $\underline{O}$, respectively. □

NOTATION **2** The expression $\underline{e}[\underline{t}]$ denotes the result of replacing the occurrence of [] in $\underline{e}$ by $\underline{t}$. This notation will also be used for more general notions of context and term. The expression $t[t_i/v_i]^n_{i=1}$ denotes the result of simultaneously replacing all occurrences of v_i in t by t_i for all $i = 1 \ldots n$. □

We have the *unique decomposition property:* For any ground term $\underline{t}$, either $\underline{t}$ is observable, or can be decomposed in exactly one way into $\underline{e}[\underline{r}]$, such that

$\underline{e}$ is a ground context, and $\underline{r}$ is a ground redex. This means that definitions and proofs concerning ground terms may split into the two cases $\underline{e}[\underline{r}]$ and $\underline{o}$; the former can be refined by the appropriate two cases of $\underline{r}$ and two cases of $\underline{e}$, if desired.

Now we can define the interpreter.[2]

DEFINITION **3** (Rewrite Interpreter.)

$$\begin{array}{lll} & \mathcal{I} : S \times \underline{T} \hookrightarrow \underline{T} & \\ (1) & \mathcal{I}[\![\ c\,\underline{t}_1 \ldots \underline{t}_n\ ]\!] & \Rightarrow\ c\,(\mathcal{I}[\![\ \underline{t}_1\ ]\!]) \ldots (\mathcal{I}[\![\ \underline{t}_n\ ]\!]) \\ (2) & \mathcal{I}[\![\ e[f\,\underline{t}_1 \ldots \underline{t}_n]\ ]\!] & \Rightarrow\ \mathcal{I}[\![\ e[\ t^f[\underline{t}_i/v_i^f]_{i=1}^n\ ]\ ]\!] \\ (3) & \mathcal{I}[\![\ e[g\,(c\,\underline{t}_{n+1} \ldots \underline{t}_{n+m})\,\underline{t}_1 \ldots \underline{t}_n]\ ]\!] & \Rightarrow\ \mathcal{I}[\![\ e[\ t^{g,c}[\underline{t}_i/v_i^{g,c}]_{i=1}^{n+m}\ ]\ ]\!] \end{array}$$

□

Thus, $\mathcal{I}[\![\ \underline{t}\ ]\!]$ is a partial algorithm which is undefined when in clause (2) or (3), the defining equation is not present in the program, and when there is no ground observable value $\underline{o}$ such that $\mathcal{I}[\![\ \underline{t}\ ]\!] \Rightarrow \underline{o}$ (*i.e.* when the result is "infinite".)

The operation of rewriting a term $e[f\,\underline{t}_1 \ldots \underline{t}_n]$ into $e[\ t^f[\underline{t}_i/v_i^f]_{i=1}^n\ ]$ conceptually proceeds in two steps: from $e[f\,\underline{t}_1 \ldots \underline{t}_n]$ to $e[\ t^f\ ]$, and from $e[\ t^f\ ]$ to $e[\ t^f[\underline{t}_i/v_i^f]_{i=1}^n\ ]$. We call the first step *unfolding* of the call to f, and the second step *binding* of the arguments in the call to f. Similarly with g-functions.

In this terminology, the meaning $\mathcal{I}[\![\ \underline{t}\ ]\!]$ of a ground term $\underline{t}$ in some program is the result obtained by repeatedly decomposing the term into the unique ground context and ground redex and then unfolding the ground redex and binding its arguments. When a ground observable value is obtained, we proceed to the constructor arguments.

[2]The notation $M_1 \ldots M_n \hookrightarrow N$ generally denotes an algorithm which expects n inputs from sets $M_1 \ldots M_n$, respectively, and yields output in N if it terminates (is defined.)

Chapter 3

Wadler's deforestation Algorithm

The deforestation algorithm was introduced in [Wad88] and is an automated application of the fold/unfold-transformation techniques described in [Bur77]. In this paper we deal with the version of deforestation presented in [Fer88].

The first section descibes the basic algorithm. The second section attempts to provide some intuition on the details of the algorithm. The basic algorithm rarely terminates. The third section describes the standard extension which makes the algorithm terminate for a large class of programs. The last section briefly considers the appropriate issues of correctness pertaining to the transformation.

3.1 The algorithm, $\mathcal{D}$

First the notions of context etc. are extended to encompass non-ground context etc.

DEFINITION **4** (Context, redex and observable.) Let e, r, o be generic variables ranging over *(lazy evaluation) contexts*, *redexes* and *observables*, respectively, as defined by the grammar:

$$\begin{array}{lcl} e & ::= & [] \mid g\ e\ t_1 \ldots t_n \\ r & ::= & f\ t_1 \ldots t_n \mid g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n \mid g\ v\ t_1 \ldots t_n \\ o & ::= & c\ t_1 \ldots t_n \mid v \end{array}$$

where t ranges over term as previously. E, R and O denote the set of all contexts, redexes and observables, respectively. □

Every ground term, context, redex is also a general term, context, redex, respectively; the unique decomposition property holds for general terms, contexts and redexes also.

Now we have the notation to define the deforestation algorithm.

DEFINITION **5** (deforestation algorithm.)

$$\mathcal{D} : S \times T \hookrightarrow T$$

$$\begin{array}{lll}
(1) & \mathcal{D}[\![\ v\ ]\!] & \Rightarrow\ v \\
(2) & \mathcal{D}[\![\ c\ t_1 \ldots t_n\ ]\!] & \Rightarrow\ c\ \mathcal{D}[\![\ t_1\ ]\!] \ldots \mathcal{D}[\![\ t_n\ ]\!] \\
(3) & \mathcal{D}[\![\ e[f\ t_1 \ldots t_n]\ ]\!] & \Rightarrow\ \mathcal{D}[\![\ e[\ t^f[t_i/v_i^f]_{i=1}^n\ ]\ ]\!] \\
(4) & \mathcal{D}[\![\ e[g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n]\ ]\!] & \Rightarrow\ \mathcal{D}[\![\ e[\ t^{g,c}[t_i/v_i^{g,c}]_{i=1}^{n+m}\ ]\ ]\!] \\
(5) & \mathcal{D}[\![\ e[g\ v\ t_1 \ldots t_n]\ ]\!] & \Rightarrow\ g'\ v\ u_1 \ldots u_k \\
 & \text{where} & \\
 & g'\ p_1^g\ u_1 \ldots u_k & =\ \mathcal{D}[\![\ e[t^{g,c_1}[t_i/v_i^{g,c_1}]_{i=1}^n]\ ]\!] \\
 & & \vdots \\
 & g'\ p_m^g\ u_1 \ldots u_k & =\ \mathcal{D}[\![\ e[t^{g,c_m}[t_i/v_i^{g,c_m}]_{i=1}^n]\ ]\!]
\end{array}$$

In clause (5) the formal parameters $u_1 \ldots u_k$ are calculated as all the variables occurring in e or one of the t_i's. Note that v may be among $u_1 \ldots u_k$. □

Note that clause (5) contains a code generation action: the term $e[gvt_1 \ldots t_n]$ is transformed into a call $g'\ v\ u_1 \ldots u_k$ to a new function. We can imagine that the where-construct is present in our language, or we can imagine that these new functions are collected somehow in a new program. We take the liberty of being imprecise on this point.

As with $\mathcal{I}$, $\mathcal{D}$ is undefined if either the required definition doesn't exist in clause (3)-(5), or no observable value can be the result. Note that the definition of $\mathcal{D}$ is exhaustive over all terms.

3.2 Chin's Producer-Consumer explanation of deforestation

As mentioned in the introduction, the purpose of the algorithm is the elimination of *intermediate data structures* from its input term. We can now make

this idea more precise.

The unique decomposition of a non-observable term into a context and a redex will always have one of the forms: $e[f t_1 \ldots t_n]$, $e[g(c t_{n+1} \ldots t_{n+m}) t_1 \ldots t_n]$, $e[g\, v\, t_1 \ldots t_n]$. Using a slightly different notation we can make this more explicit:

$$\begin{array}{rl} g_1\ (g_2\ \ldots (g_n\ [f\ t_1 \ldots t_k]\ t_1^n \ldots t_{m_n}^n) \ldots)\ t_1^1 \ldots t_{m_1}^1 & (n \geq 0) \\ g_1\ (g_2\ \ldots [g_n\ (c\ t_1 \ldots t_k)\ t_1^n \ldots t_{m_n}^n] \ldots)\ t_1^1 \ldots t_{m_1}^1 & (n > 0) \\ g_1\ (g_2\ \ldots [g_n\ v\ t_1^n \ldots t_{m_n}^n] \ldots)\ t_1^1 \ldots t_{m_1}^1 & (n \geq 0) \end{array}$$

where the terms $t_1^i \ldots t_{m_i}^i$ are arguments of g_i. In all cases, $\mathcal{D}$ unfolds the redex, in the third case after necessary *instantiations* of v to the different patterns of g's definition.

Using the terminology of Chin [Chi90] (and his later works), the idea[1] here is that the redex through a number of unfoldings evaluates to a term with an outermost constructor, *produces a constructor*. This will allow the surrounding g-function to be unfolded, *consuming* exactly the outermost constructor from the term, since patterns are one constructor deep. This latter unfolding will itself through a number of subsequent unfoldings produce a constructor allowing the next surrounding g-function to be unfolded. In this way, the constructor propagates all the way to the root of the term, and transformation then proceeds to each of the arguments of the constructor in a similar fashion.

Note that the algorithm is very similar to the interpreter of chapter 2. Actually, it is easy to see that applying $\mathcal{D}$ and $\mathcal{I}$ to a ground term yields the same result. The differences are due to the fact that the interpreter $\mathcal{I}$ gets a *ground* term and a program, whereas the deforestation algorithm $\mathcal{D}$ gets a general term and a program.

Indeed, we can imagine that $\mathcal{D}$ "thinks" as follows: I get a term t containing variables. At run-time the values for these variables will be supplied, yielding a ground term t', and thereby enough information to calculate the result of applying the interpreter to t'. I don't have those values, but let me see how much of the result I can figure out nevertheless.

When in clause (1) $\mathcal{D}$ encounters a variable, it really knows nothing about what this will be at run-time; consequently it simply returns the variable. In

[1] The following description represents the *desired* situation; the algorithm does not behave this well on all programs, as we shall see.

clauses (2)-(4), all the necessary information is present, and $\mathcal{D}$ does the same as $\mathcal{I}$ would. In clause (5), $\mathcal{D}$ does not know what the value for v will be. Therefore it does not know which definition of g to pick. A simple descision would be to do the as in clause (1), simply return $e[g\, v\, t_1 \ldots t_n]$ as the result. However, if each (or just some) of the bodies of the definitions of g evaluate to outermost constructors, then the surrounding g-function call in $e[]$ could consume these constructors, as described above; this is indeed achieved by the present clause (5).

3.3 The basic problem of Infinite Unfolding and its solution

The deforestation algorithm of the preceding section often loops infinitely.

EXAMPLE **3** Consider the following term and program:

$$\begin{array}{lcl} & & a\ (a\ zs\ ws)\ ts \\ a\ Nil\ ys & = & ys \\ a\ (Cons\ x\ xs)\ ys & = & Cons\ x\ (a\ xs\ ys) \end{array}$$

Here a is the append function, and the term is double append; it appends three lists. The inefficiency is apparent: the cons-cells of zs, will be deallocated by the inner append which will allocate new cons-cells; these cells are deallocated by the outer append, which in turn allocates new ones.

Here is what happens when we run the present deforestation algorithm on the term and program above:

$$
\begin{array}{lcl}
\mathcal{D}[\![\ a\ (a\ zs\ ws)\ ts\ ]\!] & \Rightarrow & g_1\ zs\ ws\ ts \\
\quad \text{where} & & \\
\quad g_1\ Nil\ ws\ ts & = & \mathcal{D}[\![\ a\ ws\ ts\ ]\!] \Rightarrow g_2\ ws\ ts \\
\qquad \text{where} & & \\
\qquad g_2\ Nil\ ts & = & \mathcal{D}[\![\ ts\ ]\!] \Rightarrow ts \\
\qquad g_2\ (Cons\ x\ xs)\ ts & = & \mathcal{D}[\![\ Cons\ x\ (a\ xs\ ts)\ ]\!] \\
 & \Rightarrow & Cons\ \mathcal{D}[\![\ x\ ]\!]\ \mathcal{D}[\![\ a\ xs\ ts\ ]\!] \\
 & \Rightarrow & \ldots \\
\quad g_1\ (Cons\ x\ xs)\ ws\ ts & = & \mathcal{D}[\![\ a\ (Cons\ x\ (a\ xs\ ws))\ ts\ ]\!] \\
 & \Rightarrow & \mathcal{D}[\![\ Cons\ x\ (a\ (a\ xs\ ws)\ ts)\ ]\!] \\
 & \Rightarrow & Cons\ \mathcal{D}[\![\ x\ ]\!]\ \mathcal{D}[\![\ a\ (a\ xs\ ws)\ ts\ ]\!] \\
 & \Rightarrow & \ldots
\end{array}
$$

The transformation of the original term, $a\ (a\ zs\ ws)\ ts$, gives rise to a function g_1. In the last step above, the transformation of the body of g_1 reaches the same term, $a\ (a\ xs\ ws)\ ts$ (modulo variable renaming.[2]) This will happen over and over again; the transformation process proceeds infinitely. Also, through the body of g_2, the term $a\ xs\ ts$ will be encountered over and over again.

As the example perhaps suggests, this will happen quite often when we are transforming recursive[3] functions. □

It is unsatisfactory that the algorithm doesn't terminate on a wide class of programs. A partial solution is to keep a record of all the terms the algorithm encounters. Before each transformation step of a term, say t, the algorithm checks whether t has previously been encountered. If so, the algorithm should let the result of transforming t be the same as the result obtained the first time t was encountered; and otherwise the algorithm should proceed as usual.

It will turn out in chapter 6 that we need not record *all* terms. We shall not record terms consisting only of a variable or containing an outermost

[2]That some property holds modulo variable renaming means that it holds when we identify terms which differ only in the names of variables. For instance, two terms are different modulo variable renaming if they are different even when we replace all occurrences of variables in both terms by occurrences of some variable v.

[3]Directly or mutually.

constructor. Also, when transforming the arguments of a constructor term, $c\ t_1 \ldots t_n$, the transformation of each term t_i will not be informed about the terms that have been encountered during transformation of the other t_j's. This makes the result of transformation independent of the order in which constructor arguments are transformed.

To accomodate this behaviour, the deforestation algorithm is extended as follows. First, we introduce a new function $\mathcal{F}$. Apart from the implicit program argument, this function has two arguments: a term t and a set of functions ds (a program.) The term, t, is the term that we wish to transform. The set ds provides a way of recording the terms that have previously been encountered. Each right hand side is a term that has been encountered, and the left hand side is an invented function name along with a parameter list containing the variables of the term. The $\mathcal{F}$ function always checks whether its argument term t has previously been seen by checking whether a member of ds has the term as right hand side and if so generates a call to that function (performs a *fold*); otherwise transformation should proceed as usual.

Second, $\mathcal{D}$ is changed so it introduces a *residual function call* in each of the cases of $e[r]$. This ensures that whenever $\mathcal{F}$ performs a fold then the function folded to exists in the generated program. The recursive calls of $\mathcal{D}$ are changed into calls to $\mathcal{F}$.

DEFINITION **6** (folding deforestation algorithm $\mathcal{F}$.)

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{D}' : S \times T \times S \hookrightarrow T} \\
(1) \quad \mathcal{D}'[\![\ v\ ]\!]ds & \Rightarrow & v \\
(2) \quad \mathcal{D}'[\![\ c\ t_1 \ldots t_n\ ]\!]ds & \Rightarrow & c\ (\mathcal{F}[\![\ t_1\ ]\!]ds) \ldots (\mathcal{F}[\![\ t_n\ ]\!]ds) \\
(3) \quad \mathcal{D}'[\![\ e[f\ t_1 \ldots t_n]\ ]\!]ds & \Rightarrow & f'\ u_1 \ldots u_k \\
\quad \text{where} & & \\
\quad f'\ u_1 \ldots u_k & = & \mathcal{F}[\![\ e[\ t^f[t_i/v_i^f]_{i=1}^n\ ]\ ]\!]ds \\
(4) \quad \mathcal{D}'[\![\ e[g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n]\ ]\!]ds & \Rightarrow & f'\ u_1 \ldots u_k \\
\quad \text{where} & & \\
\quad f'\ u_1 \ldots u_k & = & \mathcal{F}[\![\ e[\ t^{g,c}[t_i/v_i^{g,c}]_{i=1}^{n+m}\ ]\ ]\!]ds \\
(5) \quad \mathcal{D}'[\![\ e[g\ v\ t_1 \ldots t_n]\ ]\!] & \Rightarrow & g'\ v\ u_1 \ldots u_k \\
\quad \text{where} & & \\
\quad g'\ p_1^g\ u_1 \ldots u_k & = & \mathcal{F}[\![\ e[t^{g,c_1}[t_i/v_i^{g,c_1}]_{i=1}^n]\ ]\!]ds \\
 & \vdots & \\
\quad g'\ p_m^g\ u_1 \ldots u_k & = & \mathcal{F}[\![\ e[t^{g,c_m}[t_i/v_i^{g,c_m}]_{i=1}^n]\ ]\!]ds
\end{array}
$$

$$
\begin{array}{lllll}
 & \mathcal{F} : S \times T \times S \hookrightarrow T & & & \\
(1) & \mathcal{F}[\![\ o\ ]\!]ds & \Rightarrow & \mathcal{D}'[\![\ o\ ]\!]ds & \\
(2) & \mathcal{F}[\![\ e[r]\ ]\!]ds & \Rightarrow & h\ u_1 \ldots u_k, & \text{if } h\ u_1 \ldots u_k = e[r] \in ds \\
 & & \Rightarrow & \mathcal{D}'[\![\ e[r]\ ]\!](ds \cup \{h\ u_1 \ldots u_k = e[r]\}), & \text{otherwise}
\end{array}
$$

The membership test is to be carried out modulo variable renaming.

The formal parameters $u_1 \ldots u_k$ invented in $\mathcal{F}$ are calculated as follows. If the redex has form $g\ v\ t_1 \ldots t_n$, then $u_1 \ldots u_k$ are v followed by all the variables of $e, t_1 \ldots t_n$. Otherwise, $u_1 \ldots u_k$ are simply all the variables of $e[r]$.

The function name and parameter list invented in $\mathcal{D}$' for a given term t is chosen as the left hand side of the definition which has just been added to ds by $\mathcal{F}$. □

REMARK **1** Where our folding deforestation algorithm uses two *mutually recursive* functions $\mathcal{F}$ to fold and $\mathcal{D}'$ to (instantiate and) unfold and bind, the folding deforestation algorithm in [Fer88] uses the *composition* of two similar functions. The folding deforestation algorithm in [Chi93a] uses just one function which takes care of both folding and unfolding/binding.

We prefer the present version, which achieves an effect very similar to the one in [Fer88], because it is independent of the evaluation order of the meta-language (not the case in [Fer88]) and because it factorizes the overall problem into the two natural smaller problems (folding and unfolding/binding) (not the case in [Chi93a].) □

Let us see how this works on the example.[4]

[4] In a slightly simplified manner; the computations of ds and the intermediate applications of $\mathcal{D}'$ are not shown.

EXAMPLE **4**

$$
\begin{array}{lcl}
\mathcal{F}[\![\ a\ (a\ zs\ ws)\ ts\ ]\!] & \Rightarrow & g_1\ zs\ ws\ ts \\
\quad \text{where} & & \\
\quad g_1\ Nil\ ws\ ts & = & \mathcal{F}[\![\ a\ ws\ ts\ ]\!] \Rightarrow g_2\ ws\ ts \\
\qquad \text{where} & & \\
\qquad g_2\ Nil\ ts & = & \mathcal{F}[\![\ ts\ ]\!] \Rightarrow ts \\
\qquad g_2\ (Cons\ x\ xs)\ ts & = & \mathcal{F}[\![\ Cons\ x\ (a\ xs\ ts)\ ]\!] \\
 & \Rightarrow & Cons\ \mathcal{F}[\![\ x\ ]\!]\ \mathcal{F}[\![\ a\ xs\ ts\ ]\!] \\
\qquad\quad \text{where} & & \\
\qquad\quad \mathcal{F}[\![\ x\ ]\!] & \Rightarrow & x \\
\qquad\quad \mathcal{F}[\![\ a\ xs\ ts\ ]\!] & \Rightarrow & g_2\ xs\ ts \\
\quad g_1\ (Cons\ x\ xs)\ ws\ ts & = & \mathcal{F}[\![\ a\ (Cons\ x\ (a\ xs\ ws))\ ts\ ]\!] \\
 & \Rightarrow & f_1\ x\ xs\ ws \\
\qquad \text{where} & & \\
\qquad f_1\ x\ xs\ ws & = & \mathcal{F}[\![\ Cons\ x\ (a\ (a\ xs\ ws)\ ts)\ ]\!] \\
 & \Rightarrow & Cons\ \mathcal{F}[\![\ x\ ]\!]\ \mathcal{F}[\![\ a\ (a\ xs\ ws)\ ts\ ]\!] \\
\qquad\quad \text{where} & & \\
\qquad\quad \mathcal{F}[\![\ x\ ]\!] & \Rightarrow & x \\
\qquad\quad \mathcal{F}[\![\ a\ (a\ xs\ ws)\ ts\ ]\!] & \Rightarrow & g_2\ xs\ ws\ ts
\end{array}
$$

So the final term and program is:

$$
\begin{array}{lcl}
 & & g_1\ zs\ ws\ ts \\
\text{where} & & \\
g_1\ Nil\ ws\ ts & = & g_2\ ws\ ts \\
g_1\ (Cons\ x\ xs)\ ws\ ts & = & f_1\ x\ xs\ ws \\
f_1\ x\ xs\ ws & = & Cons\ x\ (g_1\ xs\ ws\ ts) \\
 & & \\
g_2\ Nil\ ts & = & ts \\
g_2\ (Cons\ x\ xs)\ ts & = & Cons\ x\ (g_2\ xs\ ts)
\end{array}
$$

Here $\mathcal{F}$ discovered that the repeating terms had previously been encountered.

This is almost perfect; g_2 is the append function, and g_1 is double append with the allocation and deallocation eliminated. There is only one annoying thing: the call to f_1. The term that f_1 records (the body of f_1) was

never encountered again, so it was not necessary to introduce the call to f_1. We would rather have wanted g_1 to be the following, which is obtained by unfolding the call to f_1:

$$\begin{array}{lcl} g_1\ Nil\ ws\ ts & = & g_2\ ws\ ts \\ g_1\ (Cons\ x\ xs)\ ws\ ts & = & Cons\ x\ g_1\ xs\ ts \end{array}$$

But we couldn't know that f_1 wasn't needed until we were done; we return to this problem in chapter 5. In the subsequent examples we generally only introduce residual functions which are necessary. That this does not make any significant difference is also explained in chapter 5. □

3.4 Overview of correctness issues

There are three issues of correctness for $\mathcal{F}$: preservation of operational semantics, termination, and nondegradation of efficiency.

As for the first, if we transform the body of a function in some program s, then we would like the new function to yield the same result as the original in any application. Suppose that the definitions of the new functions h introduced in the clause (3)-(5) of $\mathcal{D}$ are collected somehow in a new program s'; then desired equivalence is: if $\mathcal{F}_s[\![\ t\ ]\!] \Rightarrow t'$, then for any substitution[5] θ such that θt (and thereby $\theta t'$) is ground, we would like $\mathcal{I}_s[\![\ \theta t\ ]\!]$ and $\mathcal{I}_{s' \cup s}[\![\ \theta t'\ ]\!]$ to both loop infinitely or both evaluate to the same term.

Whether this will actually hold with the semantics as stated above is not clear. Possibly we should require preservation of semantics in a slightly weaker sense, but the matter will not be pursued in the present paper. It seems to be commonly accepted in the community that the deforestation algorithm does preserve semantics, although we have not seen a rigorous proof of that fact. A rigorous treatment of the preservation of semantics in fold/unfold transformations is given by [Cou86], [Kot??]. We have not investigated whether these results carry over to the present framework.

As for the second, we would like $\mathcal{F}$ to terminate, or at least we should know some classes of terms and programs for which it terminates. Chapter 4 describes the problems of ensuring this. Chapter 7 describes a new solution

[5]Mapping of variables to ground terms applied to terms in the usual pointwise manner. Only calls to functions present in s are allowed in these ground terms.

to these problems, and chapter 10 describes previous solutions and relates these to our new method.

As for the third, we would like the transformed program to be at least as efficient as the original program; otherwise there is hardly any point in the transformation. Chapter 5 describes the basic problems of ensuring this, along with the standard solutions.

Chapter 4

Termination problems in deforestation

The original algorithm $\mathcal{D}$ rarely terminated; therefore we introduced a new algorithm $\mathcal{F}$ which terminated more often. However, not even this algorithm is guaranteed to terminate.

The first two sections give the two archetypical examples of non-termination of $\mathcal{F}$. The third section introduces some machinery to deal with the problem.

4.1 The problem of the Accumulating Parameter

EXAMPLE **5** (The Accumulating Parameter.)

$$
\begin{array}{lcl}
 & & t\ z\ (u\ Zero) \\
 & & \\
u\ x & = & Cons\ x\ (u\ (Succ\ x)) \\
 & & \\
t\ Zero\ xs & = & Nil \\
t\ (Succ\ z)\ xs & = & t'\ xs\ z \\
 & & \\
t'\ Nil\ w & = & Nil \\
t'\ (Cons\ y\ ys)\ w & = & Cons\ y\ (t\ w\ ys)
\end{array}
$$

The u function makes an infinite list of consecutive natural numbers, starting from x, and t is a version of the well-known *take:* it takes as many elements from the list xs, as the first argument denotes. Finally, t' is an auxilary function to t, which is needed because functions admit only one pattern-matching argument. So the term computes a list consisting of the first z natural numbers.

Let us see what happens when we run $\mathcal{F}$ on this term and program.[1]

$$
\begin{array}{lcl}
\mathcal{F}[\![\, t\ z\ (u\ Zero)\ ]\!] & \Rightarrow & g_1\ z \\
\quad \text{where} & & \\
\quad g_1\ Zero & = & Nil \\
\quad g_1\ (Succ\ z) & = & \mathcal{F}[\![\, t'\ (u\ Zero)\ z\ ]\!] \\
 & \Rightarrow & \mathcal{F}[\![\, t'\ (Cons\ Zero\ (u\ (Succ\ Zero)))\ z\ ]\!] \\
 & \Rightarrow & \mathcal{F}[\![\, Cons\ Zero\ (t\ z\ (u\ (Succ\ Zero)))\ ]\!] \\
 & \Rightarrow & Cons\ Zero\ \mathcal{F}[\![\, t\ z\ (u\ (Succ\ Zero))\ ]\!] \\
\qquad \text{where} & & \\
\quad \mathcal{F}[\![\, t\ z\ (u\ (Succ\ Zero))\ ]\!] & \Rightarrow & g_2\ z \\
\qquad \text{where} & & \\
\quad g_2\ Zero & = & Nil \\
\quad g_2\ (Succ\ z) & = & \mathcal{F}[\![\, t'\ (u\ (Succ\ Zero))\ z\ ]\!] \\
 & \Rightarrow & \ldots
\end{array}
$$

Here $\mathcal{F}$ loops infinitely. The problem is that $\mathcal{F}$ encounters the larger and larger terms $t'\ (u\ Zero)\ z$, $t'\ (u\ (Succ\ Zero))\ z$, $t'\ (u\ (Succ\ (Succ\ Zero)))\ z, \ldots$ Another way of seeing the problem is that $\mathcal{F}$ encounters the succesively larger terms $t\ z\ (u\ Zero)$, $t\ z\ (u\ (Succ\ Zero))$, $t\ z\ (u\ (Succ\ (Succ\ Zero)))$, $\ldots$ The two different views identify symptoms of the same underlying problem, namely that u is called with the larger and larger arguments $Zero$, $Succ(Zero)$, $\ldots$.

Since the formal parameter of u, viz. x, is bound to larger and larger terms, Chin calls x an *accumulating parameter*. We might also in the spirit of Chin call u a *bad consumer* of its x argument, because u is not able to consume the value bound to x as quickly as it is built up in the calls to u.

Note that each of the problematic terms that are bound to x is a subterm of the term which is subsequently bound to x. □

[1] In a simplified manner; we don't introduce f functions, since these are immaterial when $\mathcal{F}$ encounters infinitely many *different* terms.

As we have indicated, our previous scheme of recording terms does not help since the terms are *different.*

4.2 The problem of the Obstructing Function call

EXAMPLE **6** (The Obstructing Function call.)

$$f\ t$$

$$\begin{array}{lcl} f\ (Leaf\ b) & = & Cons\ b\ Nil \\ f\ (Branch\ t_1\ t_2) & = & a\ (f\ t_1)\ (f\ t_2) \\ \\ a\ Nil\ ys & = & ys \\ a\ (Cons\ x\ xs)\ ys & = & Cons\ x\ (a\ xs\ ys) \end{array}$$

The f function is *flatten:* a function that turns a tree into a list, and a is the usual append function. Here is what happens when we run $\mathcal{F}$ on this term and program:

$$\begin{array}{lcl} \mathcal{F}[\![\ f\ t\ ]\!] & \Rightarrow & g_1\ t \\ \quad \text{where} & & \\ \quad g_1\ Leaf\ b & = & \mathcal{F}[\![\ Cons\ b\ Nil\ ]\!] \Rightarrow Cons\ b\ Nil \\ \quad g_1\ Branch\ t_1\ t_2 & = & \mathcal{F}[\![\ a\ (f\ t_1)\ (f\ t_2)\ ]\!] \\ & \Rightarrow & g_2\ t_1\ t_2 \\ \qquad \text{where} & & \\ \qquad g_2\ (Leaf\ b)\ t_3 & = & \ldots \\ \qquad g_2\ (Branch\ t_1\ t_2)\ t_3 & = & \mathcal{F}[\![\ a\ (a\ (f\ t_1)\ (f\ t_2))\ (f\ t_3)\ ]\!] \\ & \Rightarrow & g_3\ t_1\ t_2\ t_3 \\ & \vdots & \end{array}$$

The problem is that $\mathcal{F}$ encounters the larger and larger terms ft, $a(ft_1)ft_2$, $a\ (a\ (f\ t_1)\ f\ t_2)\ (f\ t_2)$.

We call each of the calls to f in the redex position an *obstructing function call*, since they prevent the surrounding term from ever being transformed.[2]

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

We might also in the spirit of Chin call f a *bad producer*, because it will never evaluate to a term with an outermost constructor that the surrounding a could consume.

Note that each of the problematic terms that $\mathcal{F}$ encounters appears in the redex position of the subsequent problematic term. □

4.3 The generalizing deforestation algorithm, $\mathcal{G}$

Now let us see what we can do to prevent the problems of the preceding section.

In the spirit of Section 3.3, we could in each step check whether the term contains (in some sense) a term that had previously been seen and in such case inform $\mathcal{F}$ that the term be "handled with care" since it is infinitely growing. Turchin has such a method for determining *on-line*, *i.e.* during transformation, which terms should be handled with care, for his supercompiler [Tur88]. On-line methods have also been devised for partial evaluators [Wei91].

We are, however, interested in an *off-line* strategy. Traditionally, self-applicable partial evaluators have been off-line [Bon90], [Jon91], [Jon92]. On-line evaluators have also been self-applied with some success recently but not to an extent comparable with the success of off-line partial evaluators. Moreover, as far as we know, successful automatic self-application has not been achieved for the supercompiler. We hope that our analysis ensuring termination of deforestation may evolve into a method powerful enough to make self-application of an off-line version of the supercompiler possible.

So we need to address three issues: (1) how to decide which subterms should be handled with care; (2) what it means for $\mathcal{F}$ to handle a term with care; (3) how to inform $\mathcal{F}$ that it should handle a term with care. (1) is the subject of chapter 7, and (2) and (3) are addressed below.

With intuition from the examples, there are two things that can go wrong when transforming $e[r]$. These two symptoms constitute a simplified picture of what is really going on, but will suffice as intuition for the subsequent development. (1) if $r \equiv h\ t_1 \ldots t_n$, then $e[r]$ can, in a number of steps, be transformed into $e[r']$ where $r' \equiv h\ t'_1 \ldots t'_n$ and some t'_i is a superterm of t_i

for some i. This is the phenonenom of the accumulating parameter. (2) r can in a number of steps be transformed into $e'[r]$ for some $e' \neq []$. This is the phenonenom of the obstructing function call.

We analyze each of these two problems below and consider appropriate changes to $\mathcal{F}$. We subsequently show how these changes to $\mathcal{F}$ can be implemented by adding a new function $\mathcal{G}$ that calls $\mathcal{F}$, so the changes of $\mathcal{F}$ we consider below serve only analytical purposes.

In the first case it seems natural to allow $\mathcal{F}$ to unfold h but prevent it from replacing the i'th formal parameter with the i'th actual parameter, where i is the position of the the accumulating parameter above. Surely, this will ensure that that h is not called with succesively larger arguments.

In the case where there is just one accumulating parameter, with index i, this can be achieved by redefining $\mathcal{F}$ to:[3]

$$\mathcal{F}[\![\ e[h\ t_1 \ldots t_n]\ ]\!] \Rightarrow \text{ let } v = \mathcal{F}[\![\ t_i\ ]\!] \text{ in } \mathcal{F}[\![\ e[h\ t_1 \ldots t_{i-1}\ v\ t_{i+1} \ldots t_n\ ]\!]$$

where v is a fresh variable. The step of pulling out t_i is usually called *extraction*. Specifically we say that v_i is extracted.

How are we to inform $\mathcal{F}$ that we wish the i'th parameter, or in general the i_1'th $\ldots$ i_k'th parameters, in some call to h to be extracted if the call ever ends up in a redex position? Well, to each f-function definition we assign a list of the parameters that we wish to have extracted. To the k'th clause of each g-function definition we assign a pair of lists I, I_k of parameters we wish to have extracted. I is the same for all the clauses of the g-definition and contains parameters not occurring in the patterns of the g-function. I_k contains parameters of the pattern of the k'th clause of the g-definition and is typically different for each clause of the g-function definition; this is natural since the different patterns are unrelated and may (and usually will) contain a different number of variables.

Faced with a call $g\,v\,t_1 \ldots t_n$, algorithm $\mathcal{F}$ will extract those terms among $t_1 \ldots t_n$ which have their index in I. Faced with a call $g(c_k t_{n+1} \ldots t_{n+m})t_1 \ldots t_n$, algorithm $\mathcal{F}$ will extract those terms among $t_1 \ldots t_{n+m}$ which have their index in $I \cup I_k$.

[3]The let-construct is a new subject language construct, so the right hand side of $\mathcal{F}$ contains a code generation action. The let-construct is lazy and non-recursive; in terms of our rewrite interpreter, the intended semantics is: $\mathcal{I}[\![\ \text{let } v = t \text{ in } t'\ ]\!] \Rightarrow \mathcal{I}[\![\ t'[t/v]\ ]\!]$, so let $v = t$ in t' is equivalent to $f\ t$ where f is defined as $f\ v = t'$.

This concludes the first case. Since we annotate function *definitions* rather than function *calls* with sets of parameters that are to be extracted, one might call our annotation scheme *monovariant* in analogy with the terminology for binding time analysis in partial evaluation. The annotation scheme below for the second of our cases will also be monovariant; we make some remarks on a generalization to a polyvariant, *i.e.* function *call* based, scheme in Section 9.3.

In the second case it seems natural to prevent $\mathcal{F}$ from unfolding the function call in the redex. Surely, this will prevent r from reproducing itself infinitely in the redex position. In the case where r is an obstructing function call, this can be achieved by redefining $\mathcal{F}$ to:

$$\mathcal{F}[\![\, e[r] \,]\!] \Rightarrow \text{ let } v = \mathcal{F}[\![\, r \,]\!] \text{ in } \mathcal{F}[\![\, e[v] \,]\!]$$

where v is a fresh variable. The step of pulling out r is also called *extraction*.

We can inform $\mathcal{F}$ that we wish r to be extracted as follows. If r is a call to a f-function we annotate the definition of the function with a $\ominus$. If r should not be extracted we annotate the definition with a $\oplus$. If r is a call to a g-function we assign the same symbol $\oplus$ or $\ominus$ to all the clauses of the definition.

In conclusion, before transforming our program and term, we annotate the function symbol h of every definition in the program with $\ominus$ if the calls to h in the redex position are to be extracted (*i.e.* if h is an obstructing function call) and $\oplus$ otherwise (the same symbol for all the clauses of a g-definition), and we assign a list to h of the parameters of h which are to be extracted from the calls to h in the redex position (the accumulating parameters) (in the case of a g-definition, all the lists contain the same parameters not occurring in patterns.) Note in particular that we have two means of annotation and two notions of extraction.

Gine two such annotations a_1, a_2 we say that a_2 is worse than a_1 (equivalently a_1 is better than a_2) if a_2 puts $\ominus$ on (at least) all the function definitions that a_1 puts $\ominus$ on, and every index set in a_2 is a (not necessarily strict) superset of the corresponding set in a_1. Worse annotations lead to less transformation.

The extraction mechanism can be incorporated by a new function $\mathcal{G}$ which calls $\mathcal{F}$; the recursive calls in $\mathcal{D}'$ to $\mathcal{F}$ will now be to $\mathcal{G}$ instead.

DEFINITION **7** (generalizing folding deforestation algorithm)

$\mathcal{D}'' : S \times T \times S \hookrightarrow T$

$$
\begin{array}{lll}
(1) & \mathcal{D}''[\![\ v\ ]\!]ds & \Rightarrow\ v \\
(2) & \mathcal{D}''[\![\ c\ t_1 \ldots t_n\ ]\!]ds & \Rightarrow\ c\ (\mathcal{G}[\![\ t_1\ ]\!]ds) \ldots (\mathcal{G}[\![\ t_n\ ]\!]ds) \\
(3) & \mathcal{D}''[\![\ e[f\ t_1 \ldots t_n]\ ]\!]ds & \Rightarrow\ f'\ u_1 \ldots u_k \\
 & \text{where} & \\
 & f'\ u_1 \ldots u_k & =\ \mathcal{G}[\![\ e[\ t^f[t_i/v_i^f]_{i=1}^n\ ]\ ]\!]ds \\
(4) & \mathcal{D}''[\![\ e[g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n]\ ]\!]ds & \Rightarrow\ f'\ u_1 \ldots u_k \\
 & \text{where} & \\
 & f'\ u_1 \ldots u_k & =\ \mathcal{G}[\![\ e[\ t^{g,c}[t_i/v_i^{g,c}]_{i=1}^{n+m}\ ]\ ]\!]ds \\
(5) & \mathcal{D}''[\![\ e[g\ v\ t_1 \ldots t_n]\ ]\!] & \Rightarrow\ g'\ v\ u_1 \ldots u_k \\
 & \text{where} & \\
 & g'\ p_1^g\ u_1 \ldots u_k & =\ \mathcal{G}[\![\ e[t^{g,c_1}[t_i/v_i^{g,c_1}]_{i=1}^n]\ ]\!]ds \\
 & & \vdots \\
 & g'\ p_m^g\ u_1 \ldots u_k & =\ \mathcal{G}[\![\ e[t^{g,c_m}[t_i/v_i^{g,c_m}]_{i=1}^n]\ ]\!]ds
\end{array}
$$

$\mathcal{F} : S \times T \times S \hookrightarrow T$

$$
\begin{array}{lll}
(1) & \mathcal{F}[\![\ o\ ]\!]ds & \Rightarrow\ \mathcal{D}''[\![\ o\ ]\!]ds \\
(2) & \mathcal{F}[\![\ e[r]\ ]\!]ds & \Rightarrow\ h\ u_1 \ldots u_k, \qquad \text{if } h\ u_1 \ldots u_k = e[r] \in ds \\
 & & \Rightarrow\ \mathcal{D}''[\![\ e[r]\ ]\!](ds \cup \{h\ u_1 \ldots u_k = e[r]\}), \qquad \text{otherwise}
\end{array}
$$

$\mathcal{G} : S \times T \times S \hookrightarrow T$

$$
\begin{array}{lll}
(1) & \mathcal{G}[\![\ o\ ]\!]ds & \Rightarrow\ \mathcal{F}[\![\ o\ ]\!]ds \\
(2) & \mathcal{G}[\![\ e[h^I_{\ominus}\ t_1 \ldots t_n]\ ]\!]ds & \Rightarrow\ \text{let}_{i \in I}\ v_i = \mathcal{G}[\![\ t_i\ ]\!]ds, v = \mathcal{F}[\![\ h\ u_1 \ldots u_n[v_i/u_i, t_j/u_j]_{i \in I, j \notin I}\ ]\!]ds \\
 & & \quad \text{in } \mathcal{G}[\![\ e[v]\ ]\!]ds \\
(3) & \mathcal{G}[\![\ e[h^I_{\oplus}\ t_1 \ldots t_n]\ ]\!]ds & \Rightarrow\ \text{let}_{i \in I}\ v_i = \mathcal{G}[\![\ t_i\ ]\!]ds \\
 & & \quad \text{in } \mathcal{F}[\![\ e[h\ u_1 \ldots u_n[v_i/u_i, t_j/u_j]_{i \in I, j \notin I}]\ ]\!]ds
\end{array}
$$

□

REMARK **2** In the two latter clauses of $\mathcal{G}$ we have been a little careless with the use of I cf. the description above. We could have made this explicit by splitting into appropriate cases at the expense of more syntax. □

Let us apply our new toy to the two archetypical examples.

EXAMPLE **7** In Example 5 we agreed that the problem was the fact that u was called with succesively larger arguments, so let us agree to extract x. The annotated program then is:[4]

[4]Using varable names rather than parameter positions.

$$t\ z\ (u\ Zero)$$

$$\begin{array}{lcl}
u_{\oplus}^{\{x\}}\ x & = & Cons\ x\ (u\ (Succ\ x)) \\
 & & \\
t_{\oplus}^{\{\}}\ Zero\ xs & = & Nil \\
t_{\oplus}^{\{\}}\ (Succ\ z)\ xs & = & t'\ xs\ z \\
 & & \\
t_{\oplus}'^{\{\}}\ Nil\ w & = & Nil \\
t_{\oplus}'^{\{\}}\ (Cons\ y\ ys)\ w & = & Cons\ y\ (t\ w\ ys)
\end{array}$$

Applying $\mathcal{G}$ to the term and annotated program then yields:[5]

$$\begin{array}{lcl}
\mathcal{G}[\![\ t\ z\ (u\ Zero)\ ]\!] & \Rightarrow & g_1\ z \\
\quad \text{where} & & \\
\quad g_1\ Zero & = & Nil \\
\quad g_1\ (Succ\ z) & = & \mathcal{G}[\![\ t'\ (u\ Zero)\ z\ ]\!] \\
 & \Rightarrow & \text{let } v = Zero \text{ in } \mathcal{F}[\![\ t'\ (u\ v)\ z\ ]\!] \\
 & \Rightarrow & \text{let } v = Zero \text{ in } f'\ v\ z \\
\qquad \text{where} & & \\
\qquad f'\ v\ z & = & \mathcal{G}[\![\ t'\ (Cons\ v\ (u\ (Succ\ v)))\ z\ ]\!] \\
 & \Rightarrow & \mathcal{G}[\![\ Cons\ v\ (t\ z\ (u\ (Succ\ v)))\ ]\!] \\
 & \Rightarrow & Cons\ v\ \mathcal{G}[\![\ t\ z\ (u\ (Succ\ v))\ ]\!] \\
\qquad\quad \text{where} & & \\
\qquad\quad \mathcal{G}[\![\ t\ z\ (u\ (Succ\ v))\ ]\!] & \Rightarrow & g_2\ z\ v \\
\qquad\qquad \text{where} & & \\
\qquad\qquad g_2\ Zero\ v & = & Nil \\
\qquad\qquad g_2\ (Succ\ z)\ v & = & \mathcal{G}[\![\ t'\ (u\ (Succ\ v))\ z\ ]\!] \\
 & \Rightarrow & \text{let } v' = Succ\ v \text{ in } f'\ v'\ z
\end{array}$$

[5]Omitting (most of) the intermediate applications of $\mathcal{F}$ and $\mathcal{D}''$, some of the simple applications of $\mathcal{G}$, computation of ds and unnecessary residual functions.

So the final program and term is:

$$
\begin{array}{lcl}
 & & g_1\ z \\
\text{where} & & \\
g_1\ Zero & = & Nil \\
g_1\ (Succ\ z) & = & \text{let } v = Zero \text{ in } f'\ v\ z \\
 & & \\
g_2\ Zero\ v & = & Nil \\
g_2\ (Succ\ z)\ v & = & \text{let } v' = Succ\ v \text{ in } f'\ v'\ z \\
 & & \\
f'\ v z & = & Cons\ v\ (g_2\ z\ v)
\end{array}
$$

Eliminating let's and the residual function call to f' yields the program:

$$
\begin{array}{lcl}
 & & g_1\ z \\
\text{where} & & \\
g_1\ Zero & = & Nil \\
g_1\ (Succ\ z) & = & Cons\ Zero\ (g_2\ z\ Zero) \\
 & & \\
g_2\ Zero\ v & = & Nil \\
g_2\ (Succ\ z)\ v & = & Cons\ (Succ\ v)\ (g_2\ z\ (Succ\ v))
\end{array}
$$

This is satisfactory. The call $g_2\,m\,n$ makes a list of m consecutive natural numbers starting with the number n, and $g_1\ m$ makes a list of the first m natural numbers, starting g_2 with $n = Zero$.

The problem of eliminating let's and residual function calls such as those to f' above, is discussed in chapter 5. □

EXAMPLE **8** In Example 6 we agreed that the problem was that f never evaluated to an outermost constructor, so let us agree to extract f. The annotated program then is:

$$
\begin{array}{lcl}
 & & f\ t \\
 & & \\
f^{\{\}}_{\ominus}\ (Leaf\ b) & = & Cons\ b\ Nil \\
f^{\{\}}_{\ominus}\ (Branch\ t_1\ t_2) & = & a\ (f\ t_1)\ (f\ t_2) \\
 & & \\
a^{\{\}}_{\oplus}\ Nil\ ys & = & ys \\
a^{\{\}}_{\oplus}\ (Cons\ x\ xs)\ ys & = & Cons\ x\ (a\ xs\ ys)
\end{array}
$$

Applying $\mathcal{G}$ to the term and annotated program then yields:

$$
\begin{array}{lcl}
\mathcal{G}[\![\ f\ t\ ]\!] & \Rightarrow & \text{let } v = \mathcal{F}[\![\ f\ t\ ]\!] \text{ in } v \\
 & \Rightarrow & \text{let } v = g_1\ t \text{ in } v \\
\quad \text{where} & & \\
\quad g_1\ (Leaf\ b) & = & Cons\ b\ Nil \\
\quad g_1\ (Branch\ t_1\ t_2) & = & \mathcal{G}[\![\ a\ (f\ t_1)\ (f\ t_2)\ ]\!] \\
 & \Rightarrow & \text{let } v = \mathcal{F}[\![\ f\ t_1\ ]\!] \text{ in } \mathcal{G}[\![\ a\ v\ (f\ t_2)\ ]\!] \\
 & \Rightarrow & \text{let } v = g_1\ t_1 \text{ in } g_2\ v\ t_2 \\
\qquad \text{where} & & \\
\qquad g_2\ Nil\ t_2 & = & \mathcal{G}[\![\ f\ t_2\ ]\!] \Rightarrow g_1\ t_2 \\
\qquad g_2\ (Cons\ x\ xs)\ t_2 & = & Cons\ x\ \mathcal{G}[\![\ a\ xs\ (f\ t_2)\ ]\!] \\
 & \Rightarrow & Cons\ x\ (g_2\ xs\ t_2)
\end{array}
$$

So the final program and term is:

$$
\begin{array}{lcl}
 & & \text{let } v = g_1\ t \text{ in } v \\
\text{where} & & \\
g_1\ (Leaf\ b) & = & Cons\ b\ Nil \\
g_1\ (Branch\ t_1\ t_2) & = & \text{let } v = g_1\ t_1 \text{ in } g_2\ v\ t_2 \\
 & & \\
g_2\ Nil\ t_2 & = & g_1\ t_1 \\
g_2\ (Cons\ x\ xs)\ t_2 & = & Cons\ x\ (g_2\ xs\ t_2)
\end{array}
$$

Eliminating let's yields:

$$
\begin{array}{lcl}
 & & g_1\ t \\
\text{where} & & \\
g_1\ (Leaf\ b) & = & Cons\ b\ Nil \\
g_1\ (Branch\ t_1\ t_2) & = & g_2\ (g_1\ t_1)\ t_2 \\
 & & \\
g_2\ Nil\ t_2 & = & g_1\ t_2 \\
g_2\ (Cons\ x\ xs)\ t_2 & = & Cons\ x\ (g_2\ xs\ t_2)
\end{array}
$$

This is satisfactory. The call $g_1\ t$ turns the tree t into a list, and $g_2\ l\ t$ turns the tree t into a list and appends it to the list l. □

In closing this chapter let us note that a rigorous proof that the deforestation algorithm yields an output program semantically equivalent to the original, should take the process of extraction into account.

Chapter 5

Efficiency problems in deforestation

This chapter reviews problems in ensuring that the output program from $\mathcal{G}$ is at least as efficient as the input program.

5.1 The problem of duplicated computation

EXAMPLE 9

$$\begin{array}{lcl} & & g\,z \\ g\,Nil & = & Leaf\,L \\ g\,(Cons\,x\,xs) & = & f\,(g\,xs) \\ f\,w & = & Branch\,w\,w \end{array}$$

This rather contrived program turns a list of length n into a tree of depth n. Each leaf contains a 0-ary constructor L. The result of running $\mathcal{G}$ on this program is:

$$\begin{array}{lcl} & & g'z \\ g'\,Nil & = & Leaf\,L \\ g'\,(Cons\,x\,xs) & = & Branch\,(g'\,xs)\,(g'\,xs) \end{array}$$

Now, under a call-by-value semantics, the former program has running time linear in the length of the list, while the latter has running time exponential in the length of the list. Under a call-by-name semantics like that

of our simple interpreter from Section 2.3, both programs have running-time exponential in the length of the list. However, under any lazy implementation of the call-by-name semantics, using *e.g.* graph reduction, the running times are as in the case of call-by-value.[1]

This means that the output of $\mathcal{G}$ is grossly inefficient compared to the original. Obviously, the problem is that the computation of $g'\ xs$ has been duplicated. □

In terms of our annotation mechanism this problem can be solved simply by deciding that w be extracted; then $\mathcal{G}$ will extract the computation (term) that was bound to w, and it will only be evaluated once.

How do we discover when to make a parameter unsafe as above? Well, in this case, the problems arise from the fact that f is *non-linear* in w, *i.e.* has *multiple* occurrences of w in its body. Actually, it is not hard to see that *if* the problem occurs, then there must be some function h non-linear in a variable u involved. It is also easy to see that agreeing that u be extracted solves the problem in general.

However, it may be the case that there is a non-linear function involved and yet no problem arises.

EXAMPLE **10**

$$\begin{array}{lcl} & & f\ Fst\ t \\ f\ c\ v & = & g\ c\ v\ v \\ g\ Fst\ x\ y & = & x \\ g\ Snd\ x\ y & = & y \end{array}$$

where t is some term.

Here there is no problem, because only the term bound to one of the v's will be needed. □

So, an ambitious solution to the problem would conduct an analysis of the source program p, discovering whether the problem would appear during application of $\mathcal{G}$ to p. This has been done in partial evaluation under the names of *duplication risk analysis* [Ses88] and *abstract occurrence counting*

[1] We assume that the reader is familiar with the difference between call-by-name on one hand and call-by-need or lazy evaluation on the other hand; if he isn't, he may wish to consult chapter 6 of [Bir88].

analysis [Bon90]. Incidentally, the way the information from the latter analysis is used in [Bon90] corresponds to the action taken by $\mathcal{G}$. The analysis has also been done in connection with deforestation under the name of *usage count analysis* [Ham92a], [Ham92b].

The present paper is not concerned with this particular problem, so we simply avoid the problems ad hoc.

It is worth noting that the problem treated in this section is merely a problem of *inefficiency*. In transformers of programs with call-by-value semantics, which partial evaluators usually are, there is also a problem of guaranteeing that the output of the transformer does not terminate more *often*, than the original; this can happen if the unfolding of a function discards a computation, which in turn occurs when the body of a function does not use one of its formal parameters cf. [Bon90] "Call unfolding should neither duplicate nor discard computation."

5.2 The problem of excessive residual functions

We have already noted in the preceding chapter that a residual function can be superfluous. This happens when the term which the residual function records is never encountered again. Ideally we should only introduce residual functions for terms that will be encountered more than once. This would prevent the same term from being encountered an infinite number of times. But we cannot know ahead whether a term will be encountered again.

However, *after* the transformation it is easy to figure out which functions were necessary. A very simple strategy is to put a mark on a function in ds when a fold to that function is performed during transformation. After the transformation, all calls to residual functions that didn't get a mark are unfolded. Since they didn't get a mark, no fold was ever performed to any of them and therefore this post processing cannot proceed infinitely.

Can this post processing introduce inefficiency problems like those of the preceding section? In fact, it can't. It is not hard to see that the output terms and right hand sides of new functions computed by $\mathcal{G}$ generally will conform to $O_{\mathcal{G}}$:

$$
\begin{array}{lcl}
O_{\mathcal{G}} & ::= & O_{\mathcal{F}} \mid \mathrm{let}_{i \in I}\ v_i = O_{\mathcal{G}},\ v = O_{\mathcal{F}} \text{ in } O_{\mathcal{G}} \mid \mathrm{let}_{i \in I}\ v_i = O_{\mathcal{G}} \text{ in } O_{\mathcal{F}} \\
O_{\mathcal{F}} & ::= & v \mid h\ v_1 \ldots v_n \mid c\ O_{\mathcal{G}} \ldots O_{\mathcal{G}}
\end{array}
$$

So each of the residual functions will have variables as actual parameters. Therefore, it does not introduce any inefficiency to unfold a residual function.

We could also imagine unfolding the let's. This may, however, introduce inefficiency as in the preceding section. We could make another check to see which let's could be unfolded without duplicating computation, but again, these are not the problems of the present paper, so we eliminate the let's in an ad hoc manner. Remaining let's can be expressed in terms of function calls as explained in Section 4.3.

Now it seems appropriate to address the question: does our conservative way of introducing residual functions affect the amount of inefficiency that $\mathcal{G}$ removes; that is, could we have obtained a more efficient program if we had introduced fewer residual functions.

Perhaps slightly surprisingly, the answer is *no*. To explain the point let us see how problems *could* arise in a very naive call-by-value style transformer transforming the term $g\,(f\ x)$. Let us say that f must not be unfolded. The action taken is to replace the call $f\ x$ by a call $f'\ x$ where f' is an optimized version of f. This means however that the call to g cannot be unfolded since we do not know what outermost constructor $f'\ x$ will produce, so the overall result is $g'\ (f'\ x)$ where g' is an optimized version of g.

In contrast with this our algorithm operates as follows. The result of transforming the term $g\,(f\ x)$ is $h\ x$ where h's right hand side will be the result of transforming $g\ t^f[x/v_1^f]$. If t^f has an outermost constructor the transformation algorithm can take advantage of it now, even though we introduced a residual function. The point is that the residual function is not introduced in the redex.

For efficiency of the *deforestation algorithm* it is of course desirable that the set ds be as small as possible, *i.e.* that as few residual functions as possible are introduced. A substantial line of research in mimizing the size of ds is currently being carried out in partial evaluation [Mal93]. Also, Chin [Chi93a] has some improvements over our version in this respect. We feel that these are *ad hoc* and tend to make the algorithm harder to understand and reason about and increase the amount of syntax needed to write down

the algorithm. Nevertheless, an implementation may benefit from taking them into account; we do not pursue the problem any further.

Chapter 6

Approximating $\mathcal{G}$ via $\mathcal{H}$

Chapter 3 and 4 developed three deforestation algorithms: $\mathcal{D}$, $\mathcal{F}$ and $\mathcal{G}$. $\mathcal{D}$ is the basic deforestation algorithm, $\mathcal{F}$ is a folding version of $\mathcal{D}$. $\mathcal{G}$ is a folding generalizing deforestation algorithm, the algorithm that we are really interested in. We could also have invented a non-folding generalizing deforestation algorithm $\mathcal{H}$ and then introduced $\mathcal{G}$ as a folding version of $\mathcal{H}$. Below we shall in fact be considering such a $\mathcal{H}$.

The purpose of this and the following chapters is to devise a method of finding an annotation ensuring termination of $\mathcal{G}$: for each f-function one set I and one symbol $\oplus/\ominus$; for every g-function one set I and one symbol $\oplus/\ominus$; for every clause of every g-function one set I_c. The annotation will ensure that application of $\mathcal{G}$ to the program and term with these annotations does not loop infinitely.

The first section introduces some notions to describe rigorously the terms encountered by $\mathcal{G}$, termination of $\mathcal{G}$, and some related notions. The second section then shows how the problem of finding annotations ensuring termination of $\mathcal{G}$ can be reduced to the problem of finding annotations ensuring that $\mathcal{H}$ encounters only finitely many *different* terms (modulo variable renaming.) What is needed is therefore a way of approximating merely the *set* of terms that $\mathcal{H}$ encounters (modulo variable renaming), and the problem is to guarantee that this set be finite. This task is undertaken in the two subsequent chapters.

6.1 Termination and quasi termination; computation tree

First we define the algorithm $\mathcal{H}$ alluded to above. The difference between this algorithm and $\mathcal{G}$ is that $\mathcal{G}$ folds and $\mathcal{H}$ does not.

DEFINITION **8** (generalizing deforestation algorithm.)

$$
\begin{array}{lll}
 & \mathcal{D}''' : S \times T \hookrightarrow T & \\
(1) & \mathcal{D}'''[\![\ v\ ]\!] & \Rightarrow\ v \\
(2) & \mathcal{D}'''[\![\ c\ t_1 \ldots t_n\ ]\!] & \Rightarrow\ c\ \mathcal{H}[\![\ t_1\ ]\!] \ldots \mathcal{H}[\![\ t_n\ ]\!] \\
(3) & \mathcal{D}'''[\![\ e[f\ t_1 \ldots t_n]\ ]\!] & \Rightarrow\ \mathcal{H}[\![\ e[\ t^f[t_i/v_i^f]_{i=1}^n\ ]\ ]\!] \\
(4) & \mathcal{D}'''[\![\ e[g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n]\ ]\!] & \Rightarrow\ \mathcal{H}[\![\ e[\ t^{g,c}[t_i/v_i^{g,c}]_{i=1}^{n+m}\ ]\ ]\!] \\
(5) & \mathcal{D}'''[\![\ e[g\ v\ t_1 \ldots t_n]\ ]\!] & \Rightarrow\ g'\ v\ u_1 \ldots u_k \\
 & \quad \text{where} & \\
 & \quad g'\ p_1^g\ u_1 \ldots u_k & =\ \mathcal{H}[\![\ e[t^{g,c_1}[t_i/v_i^{g,c_1}]_{i=1}^n]\ ]\!] \\
 & & \vdots \\
 & \quad g'\ p_m^g\ u_1 \ldots u_k & =\ \mathcal{H}[\![\ e[t^{g,c_m}[t_i/v_i^{g,c_m}]_{i=1}^n]\ ]\!]
\end{array}
$$

$$
\begin{array}{lll}
 & \mathcal{H} : S \times T \hookrightarrow T & \\
(1) & \mathcal{H}[\![\ o\ ]\!] & =\ \mathcal{D}'''[\![\ o\ ]\!] \\
(2) & \mathcal{H}[\![\ e[h_\ominus^I\ t_1 \ldots t_n]\ ]\!] & =\ \text{let}_{i \in I}\ v_i = \mathcal{H}[\![\ t_i\ ]\!], v = \mathcal{D}'''[\![\ h\ u_1 \ldots u_n[v_i/u_i, t_j/u_j]_{i \in I, j \notin I}\ ]\!] \\
 & & \quad \text{in}\ \mathcal{H}[\![\ e[v]\ ]\!] \\
(3) & \mathcal{H}[\![\ e[h_\oplus^I\ t_1 \ldots t_n]\ ]\!] & =\ \text{let}_{i \in I}\ v_i = \mathcal{H}[\![\ t_i\ ]\!] \\
 & & \quad \text{in}\ \mathcal{D}'''[\![\ e[h\ u_1 \ldots u_n[v_i/u_i, t_j/u_j]_{i \in I, j \notin I}]\ ]\!]
\end{array}
$$

□

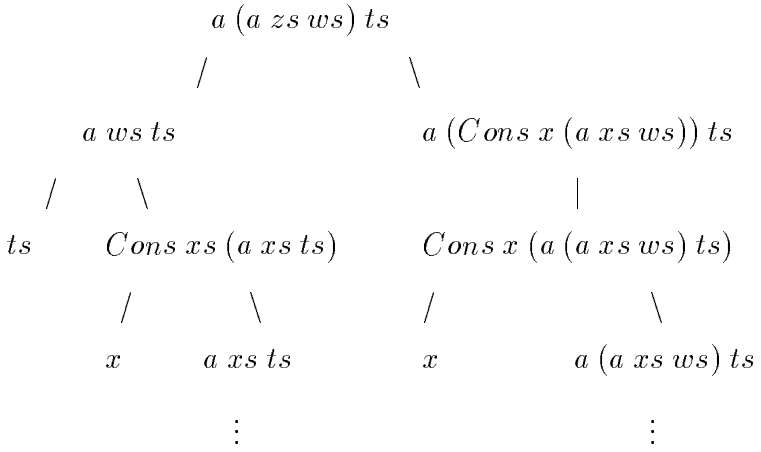
To state the relationship with respect to termination between the two generalizing algorithms $\mathcal{H}$ and $\mathcal{G}$ we need to explain what it means that $\mathcal{G}$ encounters a term. This will be explained in terms of similar notions for $\mathcal{D}$ and $\mathcal{F}$. First we explain the notions intuitively, then we make them rigorous.

Suppose that we are to apply $\mathcal{D}$ to the following term and program from Example 3 which the reader may like to consult again before moving on.

$$
\begin{array}{lll}
 & & a\ (a\ zs\ ws)\ ts \\
a\ Nil\ ys & = & ys \\
a\ (Cons\ x\ xs)\ ys & = & Cons\ x\ (a\ xs\ ys)
\end{array}
$$

The first term that occurs as argument to $\mathcal{D}$ is $a\,(a\,zs\,ws)\,ts$. We say that this is the first term that $\mathcal{D}$ *encounters*. The immediate result is a call to a new function g_1, and then $\mathcal{D}$ is applied to the two right hand sides of g_1, *viz.* $a\,ws\,ts$ and $a\,(Cons\,x\,(a\,xs\,ws))\,ts$. So $\mathcal{D}$ also encounters these two terms and we say that passing from the former to the two latter terms represents *one step* of $\mathcal{D}$. Application of $\mathcal{D}$ to the former of these two terms gives rise to a new function g_2 whose right hand sides ts and $Cons\,x\,(a\,xs\,ts)$ $\mathcal{D}$ then is applied to, *etc.*

We can write up the terms that occur as arguments to $\mathcal{D}$ in a *computation tree*, which is not necessarily finite, where each node contains a term, and the children of a node n contain the terms that arise by one step of $\mathcal{D}$ from the term in n.



We now make these notions rigorous. $\mathcal{T}_{\mathcal{D}}$ and $\mathcal{T}_{\mathcal{H}}$ compute what we have called one step of $\mathcal{D}$ and $\mathcal{H}$, respectively.[1]

[1] $\mathcal{P}$ denotes power set as usual.

DEFINITION **9** (Transition semantics of $\mathcal{D}$, $\mathcal{H}$.)

$$\mathcal{T}_{\mathcal{D}} : S \times T \hookrightarrow \mathcal{P}(T)$$

$$\begin{array}{lll}
(1) & \mathcal{T}_{\mathcal{D}}[\![\ v\ ]\!] & = \{\} \\
(2) & \mathcal{T}_{\mathcal{D}}[\![\ c\ t_1 \ldots t_n\ ]\!] & = \{t_1, \ldots, t_n\} \\
(3) & \mathcal{T}_{\mathcal{D}}[\![\ e[f\ t_1 \ldots t_n]\ ]\!] & = \{e[\ t^f[t_i/v_i^f]_{i=1}^n\ ]\} \\
(4) & \mathcal{T}_{\mathcal{D}}[\![\ e[g\ (c\ t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n]\ ]\!] & = \{e[\ t^{g,c}[t_i/v_i^{g,c}]_{i=1}^{n+m}\ ]\} \\
(5) & \mathcal{T}_{\mathcal{D}}[\![\ e[g\ v\ t_1 \ldots t_n]\ ]\!] & = \{e[\ t^{g,c_1}[t_i/v_i^{g,c_1}]_{i=1}^n], \ldots, e[\ t^{g,c_m}[t_i/v_i^{g,c_m}]_{i=1}^n\ ]\}
\end{array}$$

$$\mathcal{T}_{\mathcal{H}} : S \times T \hookrightarrow \mathcal{P}(T)$$

$$\begin{array}{lll}
(1) & \mathcal{T}_{\mathcal{H}}[\![\ o\ ]\!] & = \mathcal{T}_{\mathcal{D}}[\![\ o\ ]\!] \\
(2) & \mathcal{T}_{\mathcal{H}}[\![\ e[h^I_{\ominus}\ t_1 \ldots t_n]\ ]\!] & = \cup_{i \in I}\{t_i\} \cup \mathcal{T}_{\mathcal{D}}[\![\ h\ u_1 \ldots u_n[v_i/u_i, t_j/u_j]_{i \in I, j \notin I}\ ]\!] \cup \{e[v]\} \\
(3) & \mathcal{T}_{\mathcal{H}}[\![\ e[h^I_{\oplus}\ t_1 \ldots t_n]\ ]\!] & = \cup_{i \in I}\{t_i\} \cup \mathcal{T}_{\mathcal{D}}[\![\ e[h\ u_1 \ldots u_n[v_i/u_i, t_j/u_j]_{i \in I, j \notin I}]\ ]\!]
\end{array}$$

□

We can now describe the computation $\mathcal{H}[\![\ t_0\ ]\!]$ precisely by what we have called a computation tree.

DEFINITION **10** (Computation tree for $\mathcal{D}$, $\mathcal{H}$.) For $\mathcal{K} = \mathcal{D}$, $\mathcal{K} = \mathcal{H}$ define the tree $\mathcal{C}_{\mathcal{K}}[\![\ t_0\ ]\!]$ as follows. At level 0 there is the root containing t_0. For every node at the i'th level containing a term t, there is in $\mathcal{C}_{\mathcal{K}}[\![\ t_0\ ]\!]$ for every $t' \in \mathcal{T}_{\mathcal{K}}[\![\ t\ ]\!]$ a child at the $i+1$'th level containing t'. □

We now define the similar notions for $\mathcal{F}$ and $\mathcal{G}$.

DEFINITION **11** (Transition semantics of $\mathcal{F}$, $\mathcal{G}$.)

$$\mathcal{T}_{\mathcal{G}} : S \times T \times \mathcal{P}(T) \hookrightarrow \mathcal{P}(T)$$

$$\begin{array}{lll}
(1) & \mathcal{T}_{\mathcal{G}}[\![\ o\ ]\!]ds & = \mathcal{T}_{\mathcal{F}}[\![\ o\ ]\!]ds \\
(2) & \mathcal{T}_{\mathcal{G}}[\![\ e[h^I_{\ominus}\ t_1 \ldots t_n]\ ]\!]ds & = \cup_{i \in I}\{t_i\} \cup \mathcal{T}_{\mathcal{F}}[\![\ h\ u_1 \ldots u_n[v_i/u_i, t_j/u_j]_{i \in I, j \notin I}\ ]\!]ds \cup \{e[v]\} \\
(3) & \mathcal{T}_{\mathcal{G}}[\![\ e[h^I_{\oplus}\ t_1 \ldots t_n]\ ]\!]ds & = \cup_{i \in I}\{t_i\} \cup \mathcal{T}_{\mathcal{F}}[\![\ e[h\ u_1 \ldots u_n[v_i/u_i, t_j/u_j]_{i \in I, j \notin I}]\ ]\!]ds
\end{array}$$

$$\mathcal{T}_{\mathcal{F}} : S \times T \times \mathcal{P}(T) \hookrightarrow \mathcal{P}(T)$$

$$\begin{array}{lll}
(1) & \mathcal{T}_{\mathcal{F}}[\![\ o\ ]\!]ds & = \mathcal{T}_{\mathcal{D}}[\![\ o\ ]\!]ds \\
(2) & \mathcal{T}_{\mathcal{F}}[\![\ e[r]\ ]\!]ds & = \{\} \qquad \text{if } e[r] \in ds \\
 & & = \mathcal{T}_{\mathcal{D}}[\![\ e[r]\ ]\!](ds \cup \{e[r]\}) \qquad \text{otherwise}
\end{array}$$

□

As in algorithm $\mathcal{F}$ the membership test $\in ds$ is to be carried out modulo variable renaming.

For the computation of $\mathcal{G}[\![\ t_0\ ]\!]$ we need a tree $\mathcal{C}_\mathcal{G}[\![\ t_0\ ]\!]$ where every node contains a pair consisting of a term and a set of terms ds that has previously been encountered. The set ds corresponds to the set ds in the $\mathcal{F}$ and $\mathcal{G}$ algorithms, except that the left hand sides in ds have been left out since they are not used.

DEFINITION **12** (Computation tree for $\mathcal{F}$,$\mathcal{G}$.) Define the tree $\mathcal{C}_\mathcal{F}[\![\ t_0\ ]\!]$ as follows. At level 0 there is a node containing t_0 and the empty set (when the algorithm is started with t_0 it has not previously encountered any terms.) For every node at the i'th level with term set ds there is for every $t' \in \mathcal{T}_\mathcal{F}[\![\ t\ ]\!]ds$ a child at the $i+1$'th level containing t'. The children at the $i+1$'th level arising from clause (1) of $\mathcal{T}_\mathcal{F}$ have term set ds. The children arising from clause (2) have term set $ds \cup \{e[r]\}$.

Define the tree $\mathcal{C}_\mathcal{G}[\![\ t_0\ ]\!]$ as follows. At level 0 there is a node containing t_0 and the empty set. For every node at the i'th level with term set ds there is for every $t' \in \mathcal{T}_\mathcal{G}[\![\ t\ ]\!]ds$ a child at the $i+1$'th level containing t'. The children at the $i+1$'th level arising from clause (1) of $\mathcal{T}_\mathcal{G}$ (here clause (1) of $\mathcal{T}_\mathcal{F}$ must apply) and from $\cup_{i\in I}\{t_i\}$ and $\{e[v]\}$ in clause (2) and (3) of $\mathcal{T}_\mathcal{G}$ have term set ds. The children arising from $\mathcal{T}_\mathcal{F}[\![\ s\ ]\!]ds$ in clause (2) and (3) of $\mathcal{T}_\mathcal{G}$ (if any, clause (2) of $\mathcal{T}_\mathcal{F}$ must apply) have term set $ds \cup \{s\}$. □

By the terms encountered by $\mathcal{K}[\![\ t_0\ ]\!]$ we mean the terms occurring in a node in the tree $\mathcal{C}_\mathcal{K}[\![\ t_0\ ]\!]$. We say that $\mathcal{K}[\![\ t_0\ ]\!]$ is terminating if $\mathcal{C}_\mathcal{K}[\![\ t_0\ ]\!]$ is finite. We say that $\mathcal{K}[\![\ t_0\ ]\!]$ is quasi-terminating[2] if the nodes of $\mathcal{C}_\mathcal{K}[\![\ t_0\ ]\!]$ contain only finitely many different terms (not considering the term set components in the case of $\mathcal{F}$,$\mathcal{G}$.) Finally we say that $\mathcal{K}[\![\ t_0\ ]\!]$ is quasi-terminating modulo variable renaming, if $\mathcal{C}_\mathcal{K}[\![\ t_0\ ]\!]$ contains only finitely many different terms modulo variable renaming.

It should be noted that these are *definitions*, not assertions. They give rigorous descriptions of concepts which are vague because we haven't specified the semantics of the metalanguage. Of course, the applicability of results using our definitions above hinges on the fact that our definitions capture the intended semantics of the metalanguage. We take this for granted.

[2]This name has been used by researchers in term-rewriting [Der87] and partial evaluation [Hol91].

The reader familiar with [Hol91] may note that we have given the notion of quasi-termination a slightly new twist in that we consider quasi-termination in the *meta*language, not in the subject language. For example, when in the terminology of [Hol91] a subject term t in subject program p is quasi-terminating, we would say that $\mathcal{I}$ applied to t and p is quasi-terminating. This is not just a matter of words because the job in the subsequent sections will be to find conditions ensuring quasi-termination of $\mathcal{H}$, not $\mathcal{I}$.

6.2 Quasi-termination Theorem

THEOREM **1** *(Quasi-termination Theorem.)* $\mathcal{G}[\![\ t_0\ ]\!]$ *is terminating if* $\mathcal{H}[\![\ t_0\ ]\!]$ *is quasi-terminating modulo variable renaming.* □

PROOF: We prove that if $\mathcal{G}$ is not terminating then $\mathcal{H}$ is not quasi-terminating modulo variable renaming.

Now, if $\mathcal{G}$ is not terminating then $\mathcal{G}$ encounters infinitely many (not necessarily different) terms. Obviously, every term that $\mathcal{G}$ encounters, $\mathcal{H}$ also encounters; in particular, if $\mathcal{G}$ encounters infinitely many different terms modulo variable renaming then so does $\mathcal{H}$. So it suffices to show that if $\mathcal{G}$ encounters infinitely many terms then $\mathcal{G}$ in fact encounters infinitely many different terms modulo variable renaming.

So suppose $\mathcal{C}_{\mathcal{G}}[\![\ t_0\ ]\!]$ is infinite. By König's Lemma[3] there must be an infinite branch $(t_0, ds_0), (t_1, ds_1), \ldots$ in $\mathcal{C}_{\mathcal{G}}[\![\ t_0\ ]\!]$. Now, it is easy to see that the term of each child added by $\mathcal{T}_{\mathcal{G}}$ is strictly smaller (by the obvious measure) than its parent except possibly the children added in clause (2) or (3) of $\mathcal{T}_{\mathcal{G}}$ via clause (2) of $\mathcal{T}_{\mathcal{F}}$. So an infinite number of times t_{i+1} must have arisen from t_i via clause (2) of $\mathcal{T}_{\mathcal{F}}$. It is clear that all the infinitely many arguments of $\mathcal{T}_{\mathcal{F}}$ must be different modulo variable renaming (otherwise the branch would have ended) and *a forteriori* the corresponding parent nodes (arguments to $\mathcal{T}_{\mathcal{G}}$) must be different modulo variable renaming. □

REMARK **3** The complications of this proof arise from the fact that $\mathcal{F}$ does not recall *all* terms: not observables, and not necessarily extracted subterms.

The other direction can also be proved.

[3]See *e.g.* [Tro88].

A similar result holds betweeen $\mathcal{D}$ and $\mathcal{F}$ (by a simpler proof) but we shall not need that result. □

Notice how the proof shows the connection between our notion of quasi-termination modulo variable renaming and the folding mechanism in $\mathcal{G}$ (and $\mathcal{F}$.) In effect, every folding mechanism defines a corresponding notion of quasi-termination.

So, to ensure termination of $\mathcal{G}$ it suffices to find annotations such that $\mathcal{H}$ encounters only finitely many *different* terms (modulo variable renaming.) Therefore we need not consider the *tree* $\mathcal{C}_{\mathcal{H}}[\![\ t_0\ ]\!]$; instead we shall be considering the *set* of terms encountered by $\mathcal{H}[\![\ t_0\ ]\!]$. This could be defined as the set of terms occurring in nodes in the tree $\mathcal{C}_{\mathcal{H}}[\![\ t_0\ ]\!]$; however, the following definition is technically more convenient. It is easy to see that the two definitions are equivalent.

DEFINITION **13** (The set of terms encountered by $\mathcal{D}$,$\mathcal{H}$.) For $\mathcal{K} = \mathcal{D}$ and $\mathcal{K} = \mathcal{H}$ define $T^{\infty}_{\mathcal{K}}$ to be the set of terms that $\mathcal{K}$ encounters:

$$\begin{array}{lcl} \overline{\mathcal{T}}_{\mathcal{K}}(Q) & = & \cup_{t \in Q} \mathcal{T}_{\mathcal{K}}[\![\ t\ ]\!] \\ T^{0}_{\mathcal{K}} & = & \{t_0\} \\ T^{i+1}_{\mathcal{K}} & = & \overline{\mathcal{T}}_{\mathcal{K}}(T^{i}_{\mathcal{K}}) \\ T^{\infty}_{\mathcal{K}} & = & \cup^{\infty}_{i=0} T^{i}_{\mathcal{K}} \end{array}$$

□

REMARK **4** The dependence of $T^{\infty}_{\mathcal{K}}$ on t_0 has not been made explicit in the syntax. □

Chapter 7

Approximating $T^{\infty}_{\mathcal{H}}$

The purpose of this chapter is to devise a way of computing annotations ensuring that $T^{\infty}_{\mathcal{H}}$, the set of terms that $\mathcal{H}[\![\ t\ ]\!]$ encounters, is finite modulo variable renaming. Recall from chapter 6 that such annotations guarantee termination of $\mathcal{G}$. We do this by means of an approximation of $T^{\infty}_{\mathcal{H}}$, which in turn uses an approximation of $T^{\infty}_{\mathcal{D}}$, the set of terms that $\mathcal{D}[\![\ t\ ]\!]$ encounters. We approximate both these sets using grammars.

The overall method runs as follows. Given a program s (without annotations) and a term t.

1. Compute a grammar G approximating $T^{\infty}_{\mathcal{D}}$.
2. Look in the grammar to see whether $T^{\infty}_{\mathcal{D}}$ is infinite.
3. If so add suitable annotations to s; otherwise stop.
4. Compute a grammar G approximating $T^{\infty}_{\mathcal{H}}$.
5. Look in the grammar to see whether $T^{\infty}_{\mathcal{H}}$ is infinite.
6. If so add suitable new annotations to s and go to 4; otherwise stop and return the accumulated annotations put on s.

Section 7.1 gives grammars approximating $T^{\infty}_{\mathcal{D}}$ for the two example programs and terms from chapter 4, and attempts to provide an informal introduction to the details of the method. Section 7.2 describes a number of notions which are necessary to state the grammar approximations and prove their correctess.

The remaining sections follow the organization of the algorithm above. Section 7.3 describes the grammar approximation of $T_{\mathcal{D}}^{\infty}$ (step 1.) Section 7.4 shows how one can see in the grammar whether $T_{\mathcal{D}}^{\infty}$ is infinite (step 2.) Section 7.5 considers informally a reasonable choice of annotations given that the answer in step 2 is positive (step 3.) However, the section shows that even with these annotations put on the program we cannot be sure that $T_{\mathcal{H}}^{\infty}$ is finite. Therefore the approximation is extended to $T_{\mathcal{H}}^{\infty}$ (step 4) and the means of detecting infinity is extended accordingly (step 5.) Section 7.6 considers rigorously how annotations should be chosen (steps 3 and 6), puts together the pieces ending up with a precise formulation of the algorithm above which is then proved correct: it terminates and it outputs annotations for which $T_{\mathcal{H}}^{\infty}$ is finite. An unannotated program is a special case of an annotated program, so steps 1 to 3 in the above algorithm are special cases of steps 4 to 6; this is reflected in the algorithm in Section 7.6.

An approximation similar to the present has appeared in [Jon87]. The present method is indeed heavily inspired by that paper. Similar approximations were used in [And86] to approximate Term Rewriting Systems and in [Mog88] to handle partially static structures in partial evaluation.

We should mention that we do not consider the efficiency of the method.

7.1 Introduction and examples

In step 1 in the overall method we actually compute two grammars, a *control* grammar and a *data* grammar, which together provide the desired approximation.

EXAMPLE **11** (Grammar for the Accumulating Parameter.) Recall the program from chapter 4:

$$
\begin{array}{lcl}
 & & t\ z\ (u\ Zero) \\
 & & \\
u\ x & = & Cons\ x\ (u\ (Succ\ x)) \\
 & & \\
t\ Zero\ xs & = & Nil \\
t\ (Succ\ z)\ xs & = & t'\ xs\ z \\
 & & \\
t'\ Nil\ w & = & Nil \\
t'\ (Cons\ y\ ys)\ w & = & Cons\ y\ (t\ w\ ys)
\end{array}
$$

The first control and data grammar that will be computed for this program and term is:

$$
\begin{array}{lcl}
N^0 & \rightarrow & t\ \bullet\ (u\ Zero) \mid N^{t,Zero} \mid N^{t,Succ} \mid Nil \mid N^y \mid t\ N^w\ N^{ys} \\
N^{t,Zero} & \rightarrow & Nil \\
N^{t,Succ} & \rightarrow & t'\ N^{xs}\ \bullet \mid N^{t',Cons} \\
N^{t',Cons} & \rightarrow & Cons\ N^y\ (t\ N^w\ N^{ys}) \\
N^u & \rightarrow & Cons\ N^x\ (u\ (Succ\ N^x)) \\
N^{xs} & \rightarrow & u\ Zero \mid N^u \mid N^{ys} \\
N^w & \rightarrow & \bullet \\
N^y & \rightarrow & N^x \\
N^{ys} & \rightarrow & u\ (Succ\ N^x) \mid N^u
\end{array}
$$

$$
\begin{array}{lcl}
N^x & \rightarrow & Zero \mid Succ\ N^x \\
N^{xs} & \rightarrow & u\ Zero \mid N^{ys} \\
N^w & \rightarrow & \bullet \\
N^y & \rightarrow & N^x \\
N^{ys} & \rightarrow & u\ (Succ\ N^x)
\end{array}
$$

In the control grammar there is a nonterminal N^f for each f-function in the program, a nonterminal $N^{g,c}$ for each clause of the definition of every g-function in the program, a nonterminal N^v for every variable in the program,[1] and finally a start nonterminal N^0. In the data grammar there is

[1] Nonterminals with no productions are not shown.

a nonterminal N^v for each variable of the program. $\bullet$ informally means an arbitrary variable.

The grammar approximates the set of terms that $\mathcal{D}[\![\ t\ z\ (u\ Zero)\ ]\!]$ encounters via certain derivations[2] from N^0.

In the control grammar we are allowed to make derivations by replacing a nonterminal N with some right hand side for N, but only if N occurs in the redex position. For instance we can derive $N^0 \to^* Cons\ N^y\ (t\ N^w\ N^{ys})$ via $N^{t,Succ}$ and $N^{t',Cons}$.

In the data grammar we are allowed to make arbitrary derivations. A combined derivation is obtained by making a derivation for $N^0 \to^* t$ in the control grammar and a number of derivations $N^{v_i} \to^* t_i$ in the data grammar and then substituting t_i for N^{v_i} in t. In any derivation we may substitute any variable for $\bullet$.

The data grammar approximates the flow of data during evaluation.[3] Specifically, N^v derives all terms that are bound to v during evaluation.

The control grammar approximates the flow of control. If, during evaluation, a call to h appeared in the redex position, then the nonterminal[4] N^h has a production, and the right hand sides of N^h show what the subsequent evaluation steps of the redex yielded. If a value bound to v ended up in

[2] In the literatue of context-free grammars one usually deals with arbitrary derivation sequences where any nonterminal may be replaced by one of its right hand sides. Specifically, $\to, \to^*, \Rightarrow, \Rightarrow^*$ usually denotes one-step derivation, transitive closure of $\to$, compatible closure of $\to$ and transitive closure of $\Rightarrow$, respectively, see *e.g.* [Aho86]. We shall deal with a number of restricted notions of derivation sequences, so we shall have to deviate from the standard notation. The course taken here, which we have found to be the most convenient, is to view a grammar as a relation between the set of nonterminals and the set of (grammar) terms, *i.e.* as a subset of the Cartesian product of these two sets, and then view derivation relations as certain closure operators on such subsets. That t is derivable from N in a grammar G via some derivation mechanism d is taken, accordingly, to mean $(N, t) \in G^d$. This will be written $N \to t \in G^d$. When there is a derivation relation naturally associated with a grammar G we simply say that $N \to t$ is derivable in G.

Assuming that there are only finitely many different N for which there exists a t so that $N \in t \in G^d$, the following phrases are synonymous: G^d is infinite; there are infinitely many pairs $N \to t \in G^d$; there exists an N such that $N \to t \in G^d$ for infinitely many different t; there exists an N so that infinitely many terms are derivable from N in G (via d.)

[3] "Evaluation of term t in program p" means: "application of $\mathcal{D}$ to t in p;" we think of $\mathcal{D}$ as an interpreter.

[4] A nonterminal N^h means either a N^f or a $N^{g,c}$.

the redex position during evaluation then the nonterminal N^v has a production, and the right hand sides of N^v show what the evaluation of the redex subsequently yielded.

Given some t_0, all terms encountered by $\mathcal{D}[\![\ t_0\ ]\!]$ are derivable from N^0 using combined derivations.

Let us see if we can trace some of the evaluation in the grammar in the example. Evaluation starts with $t\,z\,(u\,Zero)$. Correspondingly, $t \bullet (u\,Zero)$ is derivable from N^0. Now $\mathcal{D}$ instantiates z to the different patterns of t, unfolds and binds the actual parameters to the formal parameters. This explains: (1) the right hand sides $N^{t,Zero}$ and $N^{t,Succ}$ of N^0 in the control grammar; (2) that we have productions for each clause of t in the control grammar; (3) that N^{xs} has right hand side $u\,Zero$ in the data grammar. Similar simulation of $\mathcal{D}$'s actions explain the remaining productions.

The problem of the Accumulating Parameter is reflected by infinity in the data grammar. Recall that the problem in Example 5 was that u was called with the larger and larger arguments $Zero$, $Succ\,Zero$, $Succ(Succ\,Zero)$, ... The formal parameter of u is x, and these terms are in fact derivable from the nonterminal N^x in the data grammar. Also recall that we noted in Example 5 that each problematic term was a subterm of the subsequent problematic term. Notice how well this is reflected by the production $N^x \to Succ\,N^x$.

In preventing infinity the idea is to extract every variable v for which N^v derives a term strictly containing N^v in the data grammar. Using this annotation and recalculating the grammars yields control and data grammars without infinity. (It will not hold in general that the first recalculated grammars contain no infinity.) □

EXAMPLE **12** (Grammar for the Obstructing Function call.)

$$f\,t$$

$$\begin{array}{lcl} f\,(Leaf\,b) & = & Cons\,b\,Nil \\ f\,(Branch\,t_1\,t_2) & = & a\,(f\,t_1)\,(f\,t_2) \end{array}$$

$$\begin{array}{lcl} a\,Nil\,ys & = & ys \\ a\,(Cons\,x\,xs)\,ys & = & Cons\,x\,(a\,xs\,ys) \end{array}$$

The first control grammar and data grammar that will be computed for this example is:

$$
\begin{array}{lcl}
N^0 & \rightarrow & f\ \bullet \mid N^{f,Leaf} \mid N^{f,Branch} \mid \bullet \mid Nil \mid N^x \mid a\ N^{xs}\ N^{zs} \mid N^{a,Nil} \\
N^{f,Leaf} & \rightarrow & Cons\ \bullet\ Nil \\
N^{f,Branch} & \rightarrow & a\ (f\ \bullet)\ (f\ \bullet) \mid a\ N^{f,Leaf}\ (f\ \bullet) \mid a\ N^{f,Branch}\ (f\ \bullet) \mid N^{a,Cons} \\
N^{a,Nil} & \rightarrow & Nil \\
N^{a,Cons} & \rightarrow & Cons\ N^x\ (a\ N^{xs}\ N^{zs}) \\
N^x & \rightarrow & \bullet \\
N^{xs} & \rightarrow & Nil
\end{array}
$$

$$
\begin{array}{lcl}
N^x & \rightarrow & \bullet \\
N^{zs} & \rightarrow & f\ \bullet \mid N^{zs}
\end{array}
$$

Let us trace the evaluation in the grammar here too. Evaluation starts with $f\ t$ which is derivable from N^0. Since f thus appears in the redex position there are productions for $N^{f,Leaf}$ and $N^{f,Branch}$, according to the different instantiations. The right hand sides of these productions show what subsequently happened to the redex. For instance, the instantiation to the pattern $Branch\ t_1\ t_2$ yielded the term $a\ (f\ t_1)\ (f\ t_2)$ signified by the right hand side $a\ (f\ \bullet)\ (f\ \bullet)$ of $N^{f,Branch}$. According to the different instantiations of t_1, this redex in turn evaluates in one step to different terms signified by the right hand sides $a\ N^{f,Leaf}\ (f\ \bullet)$ and $a\ N^{f,Branch}\ (f\ \bullet)$ of $N^{f,Branch}$.

The problem of the Obstructing Function call is reflected by infinity in the control grammar. Recall that the problem in Example 6 was that the larger and larger terms $f\,t$, $a\,(f\,t_1)\,(f\,t_2)$, $a\,(a\,(f\,t_1)\,(f\,t_2))\,(f\,t_3)\ldots$ were encountered. These terms are in fact derivable from N^0 using combined derivations (in this case using derivations only in the control grammar, but this will not hold in general.) Also recall that we noted in Example 6 that each problematic term appeared in the redex position of the subsequent problematic term. Notice how well this is reflected by the production $N^{f,Branch} \rightarrow a\ N^{f,Branch}\ (f\ \bullet)$. In preventing infinity the idea is to extract every function h for which N^h derives a term strictly containg N^h in the control grammar. Using this annotation and recalculating the grammars yields control grammar and data grammars with no infinity. □

7.2 Grammars, grammar terms and derivations

This section introduces a few fundamental concepts that are used to state the grammar construction, precisely.

DEFINITION **14** (Grammar terms, contexts, redexes and observables) Let gt, ge, gr and go be generic variables ranging over *grammar terms, grammar (lazy evaluation) contexts, grammar redexes* and *grammar observables*, respectively.[5]

$$\begin{array}{lcl} gt & ::= & \bullet \mid c\, gt_1 \ldots gt_n \mid f\, gt_1 \ldots gt_n \mid g\, gt_0 \ldots gt_n \mid N \\ ge & ::= & [] \mid g\, ge\, gt_1 \ldots gt_n \\ gr & ::= & f\, gt_1 \ldots gt_n \mid g\, (c\, gt_{n+1} \ldots gt_{n+m})\, gt_1 \ldots gt_n \mid g\, \bullet\, gt_1 \ldots gt_n \\ go & ::= & c\, gt_1 \ldots gt_n \mid \bullet \end{array}$$

GT, GE, GR and GO denotes the set of all grammar terms, contexts, redexes and observables respectively. $\mathcal{N}$ denotes the set of all nonterminals. □

As is evident, grammar terms extend ordinary terms by the presence of *nonterminals,* ranged over by N. Also, variables have been replaced by $\bullet$ which means "any variable." This in effect means that we are identifying terms which differ only in the names of variables. The following definition is convenient for making this explicit.

DEFINITION **15** (The forgetful map.) Define $\mathcal{B} : T \hookrightarrow GT$ to be the function which maps a term t to the grammar term which arises by replacing all variables in t with $\bullet$. □

The unique decomposition property is preserved in a slightly modified form: given a grammar term t, exactly one of the following cases occur: 1) t is observable; 2) t can be decomposed uniquely into e, r such that $t \equiv e[r]$; 3) t can be decomposed uniquely into e, N such that $t \equiv e[N]$. So structural inductions and definitions will use the extra case: $e[N]$.

[5] We shall often use t, e, r, o instead of gt, ge, gr, go.

DEFINITION **16** (Grammar, production.) By a *grammar*[6] we mean a subset of $\mathcal{N} \times GT$, which will usually be written $\{N_i \to gt_i\}_{i \in I}$. Each $N_i \to t_i$ is called a *production*. □

Just as the notion of context is convenient for grabing hold of the redex and denoting the unfolding of the function call, we need some notation that allows us to pick an occurrence of a nonterminal and replace it with one of the right hand sides of the nonterminal in some grammar.

Recall that grammar contexts are certain grammar terms with a hole (occurrence of []) inside. For instance, $g\,[]\,t_1 \ldots t_n$ is a context, call it e. The term $g\,t\,t_1 \ldots t_n$ which arises by replacing the hole in the context with a term t, is denoted by $e[t]$, or $(g\,[]\,t_1 \ldots t_n)[t]$. For replacement of nonterminals, we use a similar notation:

NOTATION **3** A *data context*[7] is a grammar term, containing a number of occurrences of () wherever another grammar term might have occurred. Each occurrence of () is written with a natural number index, unique within the term.

The notation for replacement is as for usual contexts. Thus, if e is the data context $(g\,t_0\,()_1\,t_1\,()_2)$, then $e(N_1)_1(N_2)_2$ is the same grammar term as $(g\,t_0\,N_1\,t_1\,N_2)$, and $e(s_1)_1(s_2)_2$ is the same term as $(g\,t_0\,s_1\,t_1\,s_2)$.

For the above and similar examples, we use the short form: $e(\overline{N})$ and $e(\overline{s})$. □

The remainder of this section reviews some derivation relations (closure operators on grammars) which are not actually computed by the grammar construction but which play a fundamental rôle nevertheless.

DEFINITION **17** (Derivation relations o, s, c, d.) Given grammars C, D.

1. C^c is the smallest grammar satisfying the closure rules:

$$\frac{N \to t \in C}{N \to t \in C^c} \qquad \frac{N \to e[N'] \in C \;\; N' \to t \in C^c}{N \to e[t] \in C^c}$$

[6]Other authors, *e.g.* [And86], use the term *tree grammar*.

[7]What we have previously called merely *contexts* will henceforth occasionally be called *control contexts*.

2. D^d is the smallest grammar satisfying the closure rules:

$$\frac{N \to t \in D}{N \to t \in D^d} \qquad \frac{N \to e(\overline{N}) \in D \;\; \overline{N} \to \overline{t} \in D^d}{N \to e(\overline{t}) \in D^d}$$

3. $< C, D >^s$ is the smallest grammar satisfying the closure rules:

$$\frac{N \to t \in C}{N \to t \in < C, D >^s} \qquad \frac{N \to e(\overline{N}) \in C \;\; \overline{N} \to \overline{t} \in D}{N \to e(\overline{t}) \in < C, D >^s}$$

where none of the () in e appear in the redex position.[8]

4. $< C, D >^o = < C^c, D^d >^s$

□

Intuitively, the c relation can simulate the replacement of a function call by the defining body of the function, and the d relation can simulate the replacement of actual parameters for formal parameters in the body of a function. The o relation can simulate both of these mechanisms. These are the derivation relations that were informally described in Section 7.1.

7.3 Approximation of $T^{\infty}_{\mathcal{D}}$

This section develops an algorithm computing a grammar that approximates the set $T^{\infty}_{\mathcal{D}}$.

The guiding principle of the development is focus on the two archetypical problems: the Accumulating Parameter and the Obstructing Function call. In the spirit of this principle, we shall distinguish firmly between the notions of *control* and *data*. This represents a significant departure from the development in [Jon87] and somewhat complicates the development. The justification is that the present development in many cases yields a better approximation.

The general idea is as follows. $\mathcal{D}$ starts with t_0, our grammar construction starts with the control grammar $\{N^0 \to \mathcal{B}[\![\ t_0\ ]\!]\}$ and empty data grammar. For each step of $\mathcal{D}$ yielding one or more new encountered terms, we perform

[8] That is e may for example not have form $g\ ()\ t_1 \ldots t_n$.

a corresponding extension of our grammars such that the resulting grammars approximate the new set of encountered terms in a sense that will be made precise. This is done by applying the two functions below to the control grammar and adjoining the results to the previous control and data grammar, respectively.[9]

DEFINITION **18**

$$
\begin{array}{lcl}
\mathcal{V}_{\mathcal{D}}[\![\ N \to \bullet\ ]\!] & = & \{\} \\
\mathcal{V}_{\mathcal{D}}[\![\ N \to c\, t_1 \ldots t_n\ ]\!] & = & \{\} \\
\mathcal{V}_{\mathcal{D}}[\![\ N \to e[f\, t_1 \ldots t_n]\ ]\!] & = & \cup_{i=1}^{n} \{N^{v_i^f} \to t_i\} \\
\mathcal{V}_{\mathcal{D}}[\![\ N \to e[g\,(c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n]\ ]\!] & = & \cup_{i=1}^{n+m} \{N^{v_i^{g,c}} \to t_i\} \\
\mathcal{V}_{\mathcal{D}}[\![\ N \to e[g\ \bullet\ t_1 \ldots t_n]\ ]\!] & = & \cup_{j=1}^{k} \cup_{i=1}^{n} \{N^{v_i^{g,c_j}} \to t_i\} \\
\mathcal{V}_{\mathcal{D}}[\![\ N \to e[N']\ ]\!] & \equiv & \{\} \\
 & & \\
\mathcal{U}_{\mathcal{D}}[\![\ N \to \bullet\ ]\!] & = & \{\} \\
\mathcal{U}_{\mathcal{D}}[\![\ N \to c\, t_1 \ldots t_n\ ]\!] & = & \{N \to t_1, \ldots, N \to t_n\},\ \text{if } N \equiv N^0 \\
 & = & \{\},\ \text{otherwise} \\
\mathcal{U}_{\mathcal{D}}[\![\ N \to e[f\, t_1 \ldots t_n]\ ]\!] & = & \{N \to e[N^f], N^f \to t^f[N^{v_i^f}/v_i^f]_{i=1}^{n}\} \\
\mathcal{U}_{\mathcal{D}}[\![\ N \to e[g\,(c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n]\ ]\!] & = & \{N \to e[N^{g,c}], N^{g,c} \to t^{g,c}[N^{v_i^{g,c}}/v_i^{g,c}]_{i=1}^{n}\ ]\} \\
\mathcal{U}_{\mathcal{D}}[\![\ N \to e[g\ \bullet\ t_1 \ldots t_n]\ ]\!] & = & \cup_{j=1}^{k} \{N \to e[N^{g,c_j}], \\
 & & N^{g,c_j} \to t^{g,c_j}[\bullet/v_i^{g,c_j}]_{i=n+1}^{m}[N^{v_i^{g,c_j}}/v_i^{g,c_j}]_{i=1}^{n}\} \\
\mathcal{U}_{\mathcal{D}}[\![\ N \to e[N']\ ]\!] & \equiv & \{\}
\end{array}
$$

□

$\mathcal{V}_{\mathcal{D}}$ and $\mathcal{U}_{\mathcal{D}}$ approximate one step of $\mathcal{D}$: if $t' \in \mathcal{T}_{\mathcal{D}}[\![\ t\ ]\!]$, then $\mathcal{V}_{\mathcal{D}}[\![\ N^0 \to \mathcal{B}[\![\ t\ ]\!]\ ]\!]$ contains all the variable bindings made in the computation of t' from t, and $\mathcal{U}_{\mathcal{D}}[\![\ N^0 \to \mathcal{B}[\![\ t\ ]\!]\ ]\!]$ contains all the unfoldings made in the computation. The unfoldings and the bindings can be combined into the term t', as we shall see.

It is not hard to see that repeatedly applying $\mathcal{U}_{\mathcal{D}}$ and $\mathcal{V}_{\mathcal{D}}$ to the productions of the control grammar, does not yield a safe approximation. For instance if we apply $\mathcal{D}$ to the term $g\,(f\,t)$, where $f\,v{=}c\,v$ (c is a constructor) then repeatedly applying $\mathcal{U}_{\mathcal{D}}$ and $\mathcal{V}_{\mathcal{D}}$ to the elements of the control grammar will yield the control grammar $\{N^0 \to gN^f, N^f \to c\,N^v\}$ and data grammar

[9] Where it occurs, k denotes the number of clauses of the suitable g-function definition.

$\{N^v \to \bullet\}$. However, $\mathcal{D}$ will encounter $g\,c\,t$ and thereby $t^{g,c}[t/v]$ which cannot be derived from the grammars by replacing nonterminals by corresponding right hand sides of productions.

So before applying our one-step extensions to the grammars we need to do some computation yielding a larger control grammar. Above we needed to calculate $N^0 \to g\,(c\,N_v)$ from the two elements of the control grammar. It will turn out in the proof of the Safety Theorem that the following operator computes a control grammar just large enough to make the approximation safe.

DEFINITION **19** Given grammar C, $C^\diamond$ is the smallest grammar satisfying the closure rules:

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

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

The flow of data may affect the flow of control during evaluation; this happens when an actual parameter occurs as the first parameter of a g-function call, and thus determines which defining clause to choose. For instance, if we apply $\mathcal{D}$ to the term $g\,(f\,(c\,t))$, where $f\,v{=}v$ then repeatedly applying $\mathcal{U}_\mathcal{D}$ and $\mathcal{V}_\mathcal{D}$ to the elements of the control grammar yields control grammar $\{N^0 \to gN^f, N^f \to N^v\}$ and data grammar $\{N^v \to c\,\bullet\}$. However, $\mathcal{D}$ will encounter $g\,(c\,t)$ and thereby $t^{g,c}[t/v^{g,c}]$, which cannot be derived from the grammars replacing nonterminals by corresponding right hand sides of productions.

So the notions of control and data cannot be kept completely seperate; the control grammar needs to be extended with productions from the data grammar. In the example above the production $N^v \to c\,\bullet$ should have been added to the control grammar. This is the purpose of the following operator, which is called f.[10]

DEFINITION **20** (Derivation relation f.) Given grammars C, D, let $C^{f(D)}$ be the smallest grammar satisfying the closure rules:

$$\frac{N \to t \in C}{N \to t \in C^{f(D)}} \qquad \frac{N \to e[N'] \in C^{f(D)} \;\; N' \to t \in D}{N \to t \in C^{f(D)}}$$

[10]This should not be confused with f used as a function name. The author admits that the use of f as the name of a closure operator is unfortunate; a symbol like + would have been better.

We also write $< C, D >^f$ for $< C^{f(D)}, D >$. □

Now we can finally define the grammar construction.

DEFINITION **21** (Grammar construction.)

$$\begin{array}{lcl} < \overline{\mathcal{U}}_{\mathcal{D}}(C), \overline{\mathcal{V}}_{\mathcal{D}}(D) > & = & < \cup_{N \to t \in C} \mathcal{U}_{\mathcal{D}}[\![N \to t]\!], \cup_{N \to t \in D} \mathcal{V}_{\mathcal{D}}[\![N \to t]\!] \\ < U^0_{\mathcal{D}}, V^0_{\mathcal{D}} > & = & < \{N^0 \to \mathcal{B}[\![t_0]\!]\}, \{\} > \\ < U^{i+1}_{\mathcal{D}}, V^{i+1}_{\mathcal{D}} > & = & < \overline{\mathcal{U}}_{\mathcal{D}}({U^i_{\mathcal{D}}}^{\diamond}) \cup U^i_{\mathcal{D}}, \overline{\mathcal{V}}_{\mathcal{D}}({U^i_{\mathcal{D}}}^{\diamond}) \cup V^i_{\mathcal{D}} >^f \\ < U^{\infty}_{\mathcal{D}}, V^{\infty}_{\mathcal{D}} > & = & < \cup_{i=0}^{\infty} U^i_{\mathcal{D}}, \cup_{i=0}^{\infty} V^i_{\mathcal{D}} > \end{array}$$

□

The reader may care to work out the two examples in Section 7.1.

As in [Jon87], we might have used just one grammar instead of two thereby avoiding some of the closure operators we have considered. However this would either reduce the accuracy compared to the present treatment or complicate the means of detecting infinity. For instance, in our treatment a value bound to a variable is only taken into account if it is ever used (as far is the approximation can see; the f operator accounts for that.) We do yet not know whether this is significant in realistic examples.[11]

We end this section by stating the desired corretness results for the grammar construction, *viz.* the *Safety Theorem* and *Termination Theorem*; they are both proved in the next chapter.

The Safety Theorem says that every term in $T^{\infty}_{\mathcal{D}}$ is derivable via o from the computed grammar modulo variable renaming.

THEOREM **2** *(Safety.)* $\forall t \in T^{\infty}_{\mathcal{D}} : \; N^0 \to \mathcal{B}[\![t]\!] \in < U^{\infty}_{\mathcal{D}}, V^{\infty}_{\mathcal{D}} >^o$ □

COROLLARY **1** *If* $N^0 \to t < U^{\infty}_{\mathcal{D}}, V^{\infty}_{\mathcal{D}} >^o$ *for only finitely many different* t *then* $T^{\infty}_{\mathcal{D}}$ *is finite modulo variable renaming.* □

PROOF: Immediate. □

The Termination Theorem says that the grammar construction can be computed finitely and that it returns finite grammars.

[11] The author's supervisor has remarked that it probably is.

THEOREM **3** *(Termination.)* $\exists j : < \cup_{i=0}^{j} U_{\mathcal{H}}^i, \cup_{i=0}^{j} V_{\mathcal{H}}^i > = < \cup_{i=0}^{\infty} U_{\mathcal{H}}^i, \cup_{i=0}^{\infty} V_{\mathcal{H}}^i >$ □

COROLLARY **2** $\cup_{i=0}^{\infty} U_{\mathcal{H}}^i, \cup_{i=0}^{\infty} V_{\mathcal{H}}^i$ *are finite and can be computed by in a finite number of finite steps.* □

PROOF: Since we start out with finite grammars and each step adds finitely many productions, the Termination Theorem implies that $< \cup_{i=0}^{\infty} U_{\mathcal{H}}^i, \cup_{i=0}^{\infty} V_{\mathcal{H}}^{\infty} >$ is a pair of finite grammars. These may be computed in a finite number of steps simply by checking after each step whether the step added any new productions to the union of the previous grammars. If not, the computation should terminate; otherwise it should proceed. Each step can also be computed finitely; in computing $C^{\diamond}$ one must take care to avoid cycles $N_1 \to N_2 \to \ldots \to N_i \to N_1$, but this is easily done. □

7.4 Detecting infinity

Now we aim to devise a means of figuring out whether infinitely many terms are derivable from N^0 in the computed grammar via o.

DEFINITION **22** Given $< C, D >$. We say that a nonterminal N is *D-reachable* if $N^0 \to e(N) \in C^c$ for some e with () not in the redex position. We say that a nonterminal N is *C-reachable* if $N^0 \to e[N] \in C^c$. □

Intuitively N^v is D-reachable if there is some derivation $N^0 \to t \in < C, D >^o$ which uses a production $N^v \to t' \in D$. Similarly, N^h (or N^v) is C-reachable if there is some derivation $N^0 \to t \; in \; < C, D >^o$ which uses a production $N^h \to t'$. Note that it is possible that D^d is infinite and yet $N^0 \to t \in < C, D >^o$ for only finitely many t,
eg if only finitely many nonterminals in D (and thereby D^d) are D-reachable.

The following Theorem is similar to the *Pumping Lemma* known from the theory of context-free grammars, see *e.g.* [Har78].[12]

[12] An even closer correspondence with the traditional Pumping Lemma is obtained by replacing "t" in $(*)$ by "*ground* t" where ground means free of nonterminals, and adding corresponding conditions in (1) and (2). Actually, the Theorem still holds if make the

THEOREM **4** *(Infinity Detection Theorem for $< U^\infty_{\mathcal{D}}, V^\infty_{\mathcal{D}} >$.) Given program p and term t_0. Let $C = U^\infty_{\mathcal{D}}$ and $D = V^\infty_{\mathcal{D}}$. Then*

($*$) $N^0 \to t \in\!< C, D >^o$ *for infinitely many different* t

iff

(1) *there exists C-reachable* N^h, $e \neq []$ *such that* $N^h \to e[N^h] \in C^c$; *or*
(2) *there exists D-reachable* N^v, $e \neq ()$ *such that* $N^v \to e(N^v) \in D^d$.

□

PROOF: The upwards direction is trivial. To show the downwards direction we first show the intermediate assertion, that ($*$) implies that

($1'$) there exists C-reachable N, $e \neq []$ such that $N \to e[N] \in C^c$; or
($2'$) there exists D-reachable N^v, $e, \neq ()$ such that $N^v \to e(N^v) \in D^d$.

The difference here from the assertion we wish to prove is that N in ($1'$) can be a nonterminal corresponding to a variable. We show this by contraposition, *i.e.* we show $\neg(1') \wedge \neg(2') \Rightarrow \neg(*)$.

By the Termination Theorem C and D are finite. From $\neg(1')$ it is therefore easy to find an upper bound for the size of any t with $N^0 \to t \in C^c$ (the number of nonterminals times the largest right hand side.) Since these t's are build from a finite alphabet of nonterminals, variable, constructor and function names this means $N^0 \to t \in C^c$ for only finitely many different t.

Similarly we can from $\neg(2')$ deduce that if $N^0 \to e(N^v) \in C^c$, where $()$ is not in the redex position, then $N^v \to t \in D^d$ for only finitely many t. Now the deductions from $\neg(1')$ and $\neg(2')$ together show that there can only be finitely many t with $N^0 \to t \in\!< C, D >^o$ which is the negation of ($*$), as desired.

To prove the overall assertion it suffices to show that if $N^v \to e[N^v] \in C^c$ for some $N^v, e \neq []$, then either $N^v \to e[N^v] \in D^d$ or there exists a nonterminal N^h and $e' \neq []$ such that $N^h \to e'[N^h] \in C^c$.

This is proved as follows. First prove by induction on i that if $N^v \to e[N^v] \in (U^i_{\mathcal{D}})^c$ for some $N^v, e \neq []$, then either $N^v \to e[N^v] \in (V^i_{\mathcal{D}})^d$ or there

change in ($*$) but none in (1), (2). For the downwards direction this is obvious since we have strengthened the hypothesis. For the upwards direction one applies the definition of $\mathcal{U}_{\mathcal{D}}$ and $\mathcal{V}_{\mathcal{D}}$ to show that the extra conditions in (1), (2) automatically hold (so one does not need to add them as conditions.) We do not pursue the details.

exists a nonterminal N^h and $e' \neq []$ such that $N^h \to e'[N^h] \in (U^i_{\mathcal{D}})^c$. Now, if $N^v \to e[N^v] \in (U^\infty_{\mathcal{D}})^c$, then $N^v \to e[N^v] \in (U^i_{\mathcal{D}})^c$ for some i (the derivation uses only finitely many productions each lying in some $U^j_{\mathcal{D}}$; take i as the largest j.) Then *e.g.* $N^v \to e[N^v] \in (V^i_{\mathcal{D}})^d \subseteq (V^\infty_{\mathcal{D}})$, and similarly with the other possibility. □

Notice the very close correspondence between on one hand the two criteria of infinity in the grammar, $N^h \to e[N^h]$, $N^v \to e(N^v)$, and on the other hand the problems of the Accumulating Parameter and the Obstructing Function call. The factorization of the derivation relation o and the approximating grammar into data and control parts has been done to achieve this similarity.

COROLLARY **3** *Given program p and term t. Let* $C = U^\infty_{\mathcal{D}}$ *and* $D = V^\infty_{\mathcal{D}}$. *It is decidable whether* $N^0 \to t \in < C, D >^o$ *for infinitely many different t.* □

PROOF: By the Infinity Detection Theorem it suffices to find a procedure that given $< C, D >$ can discover whether

(1) there exists C-reachable N^h, $e \neq []$ such that $N^h \to e[N^h] \in C^c$; or
(2) there exists D-reachable N^v, $e \neq ()$ such that $N^v \to e(N^v) \in D^d$.

First make a graph G with four kinds of edges: *C-edges*, *D-edges*, *CR-edges* and *DR-edges*. The nodes of G are the nonterminals of C and D. Whenever $N \to e[N'] \in C$, make a C-edge from N to N' in G, and whenever $N \to e(N') \in D$, make a D-edge from N to N' in G. For CR-edges proceed as follows. For every $N^0 \to e[N'] \in C$ add a CR-edge from N^0 to N'. Whenever there is a C-edge from N to N', add for every CR-edge from N' to some N'' a CR-edge from N to N''. For DR-edges proceed as follows. For every $N \to t \in C$ add a DR-edge from N to every nonterminal in t except the one in the redex position, if any. Whenever there is a C-edge from N to N', add for every DR-edge from N' to some N'' a DR-edge from N to N''.

Now the problem is whether the graph contains a C-cycle along which $e \neq []$ starting and ending in a nonterminal with a CR-edge from N^0, or a D-cycle along which $e \neq ()$ starting and ending in a nonterminal which has an DR-edge from N^0. This is clearly decidable. □

Recall that by the Corollary to the Safety Theorem, if $N^0 \to < U^\infty_{\mathcal{D}}, V^\infty_{\mathcal{D}} >^o$ for only finitely many different terms then $T^\infty_{\mathcal{D}}$ is finite modulo variable renaming. By the preceding Theorem and Corollary we now have a way of

checking whether the antecedent is true or false. If it is true, we are done; if it is false we do not know whether $T^{\infty}_{\mathcal{D}}$ is finite modulo variable renaming or not, so we shall have to assume that it is. In the latter situation what do we do? The next section proceeds with that question.

7.5 Extension to $T^{\infty}_{\mathcal{H}}$

The basic response to an infinite $T^{\infty}_{\mathcal{D}}$ is of course to extract (1) every h for which $N^h \to e[N^h] \in C^c$ where N^h is C-reachable and $e \neq []$, and (2) every variable v for which $N^v \to e(N^v) \in D^d$ where N^v is D-reachable and $e \neq ()$. However, this by itself does not ensure that $T^{\infty}_{\mathcal{H}}$ is finite modulo variable renaming (*i.e.* that $\mathcal{G}$ terminates.)

EXAMPLE **13** Let g be the flatten function which illustrated the problem of the Obstructing Function call in the preceding chapter. Now consider the program and term:

$$\begin{array}{rcl} & & g\,(f\,u) \\ f\,v & = & f\,(f\,v) \end{array}$$

The data grammar computed from this program contains only $N^v \to f\,N^v$, and the control grammar consists of the following productions:

$$\begin{array}{rcl} N^0 & \to & g\,(f\,\bullet) \mid g\,N^f \\ N^f & \to & f\,(f\,N^v) \mid N^f \\ N^v & \to & f\,N^v \mid N^f \mid N^v \end{array}$$

Notice that no information concerning g is present.

So calls to f in a redex position are to be extracted and arguments to f are to be extracted. If we run the generalizing deforestation on the program and term with these informations, the following happens:

$$\mathcal{G}[\![\, g\,(f\,u)\,]\!] \;=\; \text{let } x = \mathcal{G}[\![\, u\,]\!], y = \mathcal{D}[\![\, f\,x\,]\!] \text{ in } \mathcal{G}[\![\, g\,y\,]\!]$$

But now the transformation of $g\,y$ will proceed infinitely. The grammar construction never found out anything about g because f never evaluated to an outermost constructor or a variable, but now we feed a variable as argument of g. In short, the problem is that the effect of extraction has not been taken into account in the grammar construction. □

There are three ways to deal with this. First, we could simply chicken out and change the generalized deforestation algorithm in such a way that it doesn't transform the extracted subexpressions, but simply extracts them and then leaves them untransformed.

Second, in each step of $\mathcal{U}_\mathcal{D}$ and $\mathcal{V}_\mathcal{D}$ we could always add some extra productions such as $N^f \to v$ as above. Possibly we should add productions $N^f \to c\,t_1 \ldots t_n$ for each constructor of a pattern of the function immediately surrounding f in the context, above for the g function. This idea is not pursued further.

Third, and theoretically more pleasing: devise a generalizing grammar construction that approximates the generalizing deforestation algorithm, or, more accurately, that approximates $T^\infty_\mathcal{H}$. The remainder of this section gives the details.

Extend the grammar construction:

DEFINITION **23** (generalizing grammar construction.)

$$
\begin{array}{lcl}
\mathcal{U}_\mathcal{H}[\![\ N \to o\ ]\!] & = & \mathcal{U}_\mathcal{D}[\![\ N \to o\ ]\!] \\
\mathcal{U}_\mathcal{H}[\![\ N \to e[h^I_\ominus\ t_1 \ldots t_n]\ ]\!] & = & \cup_{i\in I}\{N^0 \to t_i\} \cup \mathcal{U}_\mathcal{D}[\![\ N^0 \to h\ u_1 \ldots u_n[\bullet/u_i, t_j/u_j]_{i\in I, j\notin I}\ ]\!] \\
 & & \cup\{N \to e[\bullet]\} \\
\mathcal{U}_\mathcal{H}[\![\ N \to e[h^I_\oplus\ t_1 \ldots t_n]\ ]\!] & = & \cup_{i\in I}\{N^0 \to t_i\} \cup \mathcal{U}_\mathcal{D}[\![\ N \to e[h\ u_1 \ldots u_n[\bullet/u_i, t_j/u_j]_{i\in I, j\notin I}]\ ]\!]
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{V}_\mathcal{H}[\![\ N \to o\ ]\!] & = & \mathcal{V}_\mathcal{D}[\![\ N \to o\ ]\!] \\
\mathcal{V}_\mathcal{H}[\![\ N \to e[h^I_\ominus\ t_1 \ldots t_n]\ ]\!] & = & \mathcal{V}_\mathcal{D}[\![\ N^0 \to h\ u_1 \ldots u_n[\bullet/u_i, t_j/u_j]_{i\in I, j\notin I}\ ]\!] \\
\mathcal{V}_\mathcal{H}[\![\ N \to e[h^I_\oplus\ t_1 \ldots t_n]\ ]\!] & = & \mathcal{V}_\mathcal{D}[\![\ N \to e[h\ u_1 \ldots u_n[\bullet/u_i, t_j/u_j]_{i\in I, j\notin I}]\ ]\!]
\end{array}
$$

□

DEFINITION **24**

$$
\begin{array}{lcl}
< \overline{\mathcal{U}}_\mathcal{H}(C), \overline{\mathcal{V}}_\mathcal{H}(D) > & = & < \cup_{N\to t\in C}\mathcal{U}_\mathcal{H}[\![\ N \to t\ ]\!], \cup_{N\to t\in D}\mathcal{V}_\mathcal{H}[\![\ N \to t\ ]\!] > \\
< U^0_\mathcal{H}, V^0_\mathcal{H} > & = & < \{N^0 \to \mathcal{B}[\![\ t_0\ ]\!]\}, \{\} > \\
< U^{i+1}_\mathcal{H}, V^{i+1}_\mathcal{H} > & = & < \overline{\mathcal{U}}_\mathcal{H}((U^i_\mathcal{H})^\diamond) \cup U^i_\mathcal{H}, \overline{\mathcal{V}}_\mathcal{H}((V^i_\mathcal{H})^\diamond) \cup V^i_\mathcal{H} >^f \\
< U^\infty_\mathcal{H}, V^\infty_\mathcal{H} > & = & < \cup^\infty_{i=0} U^i_\mathcal{H}, \cup^\infty_{i=0} V^i_\mathcal{H} >
\end{array}
$$

□

We then have the generalized Safety Theorem and Termination Theorem which are obtained be replacing $\mathcal{D}$ with $\mathcal{H}$ everywhere in the original Theorems. The Corollaries to both Theorems hold similarly. The generalizations of the Theorem and Corollary from the preceding section obtained by replacing $\mathcal{D}$ by $\mathcal{H}$ are also true by the same proofs.

7.6 Breaking infinity; putting together the pieces

The only missing piece is provided by the following theorem.

THEOREM **5** *(Infinity Breaking Theorem.) If h is marked $\ominus$ then there is no $e \neq []$ such that $N^h \rightarrow e[N^h] \in U^\infty_{\mathcal{H}}$. If v has its index in the set of the function of which it is a formal parameter, then there is no $e \neq ()$ such that $N^v \rightarrow e(N^v) \in V^\infty_{\mathcal{H}}$.* $\Box$

PROOF: Observe that (1) the N^h described above is only introduced in the right hand sides of productions of the form $N^0 \rightarrow N^h$, and N^0 is never introduced in right hand sides; and (2) the N^v are only introduced on left hand sides of productions of the form $N^v \rightarrow \bullet$. $\Box$

THEOREM **6** *For every term t_0 and unannotated program the following algorithm terminates with a set of annotations for which $T^\infty_{\mathcal{H}}$ is finite modulo variable renaming (and hence $\mathcal{G}[\![\ t_0\ ]\!]$ terminates by the Quasi-termination Theorem.)*

1. *Let every h be annotated with $\oplus$ and let all the index sets be empty (this is the* trivial annotation.[13]*)*
2. *calculate $< C, D >=< U^\infty_{\mathcal{H}}, V^\infty_{\mathcal{H}} >$.*
3. *annotate with $\ominus$ every h for which there exists C-reachable N^h, $e \neq []$ such that $N^h \rightarrow e[N^h] \in C^c$. Put in the set of the function of which v is a formal parameter v's index, whenever there exists D-reachable N^v, $e \neq ()$ such that $N^v \rightarrow e(N^v) \in D^d$.*
4. *if step 3 yielded no new annotations stop; otherwise go to step 2.*

$\Box$

[13]Note that $\mathcal{H}$ ($T^\infty_{\mathcal{H}}$) applied to the trivially annotated program yields the same as $\mathcal{D}$ ($T^\infty_{\mathcal{D}}$) to the unannotated program.

PROOF: Since step 2 always terminates by the Corollary to the Termination Theorem, and since the annotations can only get worse a finite number of times, the algorithm terminates.

In the last grammar computed in the sequence above, only finitely many different terms are derivable from N^0: if infinitely many terms were derivable from N^0, then by the Infinity Detection Theorem for $\mathcal{H}$, either a nonterminal N^h would derive itself in a nonempty control context or a nonterminal N^v would derive itself in a nonempty data context. By the Infinity Breaking Theorem, such a nonterminal cannot be a N^h for a h already with $\ominus$ or a N^v for a v already in the list of parameters to be extracted, but this contradicts the fact that the last step did not yield any new annotations.

Since only finitely many different terms are derivable from N^0 in the grammar, $T^{\infty}_{\mathcal{H}}$ is finite modulo variable renaming by the Corollary to the Safety Theorem. □

This concludes the method.

Chapter 8

Correctness of the approximation of $T_{\mathcal{H}}^{\infty}$

This chapter proves the correctness of the grammar construction. Section 8.1 and 8.2 prove the Safety Theorem and Termination Theorem, respectively, which were quoted in the preceding chapter.

8.1 Safety

The purpose of this section is to prove the Safety Theorem, which was quoted in the preceding chapter:

$$\forall t \in T_{\mathcal{H}}^{\infty} : \quad N^0 \rightarrow \mathcal{B}[\![\, t \,]\!] \in < U_{\mathcal{H}}^{\infty}, V_{\mathcal{H}}^{\infty} >^{o}$$

First we prove the similar result for $\mathcal{D}$, then we extend that result to $\mathcal{H}$. Both these results are proved using two kinds of properties: (1) properties about the derivation relations d, c, s, o; (2) properties about the $\mathcal{U}$ and $\mathcal{V}$ functions.

The overall proof strategy isolates a few properties about $\mathcal{U}$ and $\mathcal{V}$ in two lemmas, for later reference called the *basic* lemmas. The remaining properties of $\mathcal{U}$ and $\mathcal{V}$ follow from the basic lemmas and properties of the derivation relations. This means that in generalizing our results from $\mathcal{D}$ to $\mathcal{H}$ we need only reprove the basic lemmas.

First we review the properties of the derivation relations. The two following lemmas state what we would expect.

LEMMA **1** *(Monotonicity of derivation relations.) Suppose* $X \subseteq Y$ *and* $A \subseteq B$. *Then*

$$\begin{array}{lllll} (1) & X^{f(A)} & \subseteq & Y^{f(B)} \\ (2) & A^c & \subseteq & B^c \\ (3) & A^d & \subseteq & B^d \\ (4) & < X, A >^s & \subseteq & < Y, B >^s \end{array}$$

□

PROOF: Easy. □

LEMMA **2** *(Idempotency of derivation relations.)*

$$\begin{array}{lllll} (1) & A^{cc} & = & A^c \\ (2) & A^{dd} & = & A^d \\ (3) & X^{f(B)f(B)} & = & X^{f(B)} \\ (4) & (A \cup B^c)^c & = & (A \cup B)^c \\ (5) & (A \cup B^d)^d & = & (A \cup B)^d \end{array}$$

□

PROOF: (1), (2), (3) are obvious. (4) follows from (1) and Lemma 1 (2). (5) follows from (2) and Lemma 1 (3). □

The following lemma shows how the different derivation relations commute.

LEMMA **3** *(Commutativity of derivation relations.)*

$$\begin{array}{lllll} (1) & < X^{f(Y)}, Y >^{s\, f(Y)} & \subseteq & < X^{f(Y)}, Y >^s \\ (2) & << X, Y >^s, Y >^s & \subseteq & < X, Y >^s \\ (3) & X^{c\, f(D)} & \subseteq & X^{f(D)\, c} \\ (4) & < X, Y >^{sc} & \subseteq & < X^c, Y >^s \\ (5) & X^{f(D^d)} & \subseteq & < X^{f(D)}, D^d >^s \end{array}$$

□

PROOF:

(1): By induction on the definition of the f-operator. Let $N \to t \in < X^{f(Y)}, Y >^{s\, f(Y)}$. This can happen in two ways by the definition of f. First, if $N \to t \in < X^{f(Y)}, Y >^{s}$ then we have the desired directly. The other possibility is $N \to t \in Y$ and $N' \to e[N] \in < X^{f(Y)}, Y >^{s\, f(Y)}$ for some e, N'. By induction hypothesis $N' \to e[N] \in < X^{f(Y)}, Y >^{s}$. So $e[N] \equiv e'(\overline{s})[N]$ where $N' \to e(\overline{K})[N] \in X^{f(Y)}$ and $\overline{K} \to \overline{s} \in Y$. The former yields $N \to t \in X^{f(Y)}$ and thereby $N \to t \in < X^{f(Y)}, Y >^{s}$ as desired.

(2): Let $N \to t \in << X, Y >^{s}, Y >^{s}$. So $t \equiv e(\overline{s})$ where $N \to e(\overline{K}) \in < X, Y >^{s}$ and $\overline{K} \to \overline{s} \in Y$. So $e(\overline{K}) \equiv e'(\overline{u})(\overline{K})$ where $N \to e'(\overline{L})(\overline{K}) \in X$ and $\overline{L} \to u \in Y$. Now we have $N \to e'(\overline{u})(\overline{s}) \in < X, Y >^{s}$ as desired.

(3): By induction on the definition of the f-operator. Let $M \to t \in X^{c\, f(D)}$. If $M \to t \in X^{c}$ the desired follows from Lemma 1. If $M \to t \in D$ and $N_0 \to e[M] \in X^{c\, f(D)}$ for some e, N_0 then by induction hypothesis $N_0 \to e[M] \in X^{f(D)\, c}$. So $e[M] \equiv e_1[e_2 \ldots e_{k+1}[M] \ldots]$ where for $i = 0 \ldots k-1$, $N_i \to e_{i+1}[N_{i+1}], N_k \to e_{k+1}[M] \in X^{f(D)}$. Then also $M \to t \in X^{f(D)}$.

(4): Let $N_0 \to t \in < X, Y >^{sc}$ *i.e.* $t \equiv e_1[e_2 \ldots e_k[t'] \ldots]$ where for $i = 0 \ldots k-1$, $N_i \to e_{i+1}[N_{i+1}], N_k \to t' \in < X, Y >^{s}$. So for $i = 0 \ldots k-1$, $e_{i+1}[N_{i+1}] \equiv e'_{i+1}(\overline{t}_{i+1})[N_{i+1}]$ where $N_i \to e'_{i+1}(\overline{K}_{i+1})[N_{i+1}] \in X$ and $\overline{K}_{i+1} \to \overline{t}_{i+1}$, and $t' \equiv t''(\overline{u})$ where $N_k \to t''(\overline{L}) \in X$ and $\overline{L} \to \overline{u} \in Y$. So $N \to e'_1(\overline{K}_1)[e'_2(\overline{K}_2) \ldots e'_k(\overline{K}_k)[t''(\overline{L})] \ldots] \in X^{c}$ and thereby the desired $N \to e'_1(\overline{t}_1)[e'_2(\overline{t}_2) \ldots e'_k(\overline{t}_k)[t'(\overline{u})] \ldots] \in < X^{c}, Y >^{s}$

(5): By induction on the definition of the f-operator. Let $N \to t \in X^{f(D^d)}$. If $N \to t \in X$ the desired follows using Lemma 1. If $N \to t \in D^d$ and $N' \to e[N] \in X^{f(D^d)}$ for some e, N' then by induction $N' \to e[N] \in < X^{f(D)}, D^d >^{s}$. So $e[N] \equiv e'(\overline{u})[N]$ where $N' \to e'(\overline{K})[N] \in X^{f(D)}$ and $\overline{K} \to \overline{u} \in D^d$. Since $t \equiv e''(\overline{v})$ where $N \to e''(\overline{L}) \in D$ and $\overline{L} \to \overline{v} \in D^d$, we also have $N \to e''(\overline{L}) \in X^{f(D)}$ and thereby $N \to e''(\overline{v}) \in < X^{f(D)}, D^d >^{s}$. □

There is nothing inherently interesting in the following corollary; but it will be needed later.

COROLLARY **4** $<< X^c, Y^d >^{fs}, Y^d >^{fo} \subseteq < X, Y >^{fo}$. □

PROOF: Lemma 1 is used in each step; the other results used are mentioned

in each step.

$$
\begin{array}{lll}
<< X^c, Y^d >^{fs}, Y^d >^{fo} & \equiv & \\
<< X^{c\,f(Y^d)}, Y^d >^{s\,f(Y^d)}, Y^d >^{o} & \subseteq & \text{by Lemma 3(1), Lemma 2(3)} \\
<< X^{c\,f(Y^d)}, Y^d >^{s}, Y^d >^{o} & \subseteq & \text{by Lemma 3(5)} \\
<<< X^{c\,f(Y)}, Y^d >^{s}, Y^d >^{s}, Y^d >^{o} & \subseteq & \text{by Lemma 3(2)} \\
<< X^{c\,f(Y)}, Y^d >^{s}, Y^d >^{o} & \equiv & \\
<< X^{c\,f(Y)}, Y^d >^{sc}, Y^{dd} >^{s} & \subseteq & \text{by Lemma 3(4),Lemma 2(2)} \\
<< X^{c\,f(Y)\,c}, Y^d >^{s}, Y^d >^{s} & \subseteq & \text{by Lemma 3(2,3), Lemma 2(1)} \\
< X^{f(Y)\,c}, Y^d >^{s} & \equiv & \\
< X, Y >^{fo} & &
\end{array}
$$

□

REMARK **5** The converse inclusion is easy to show using Lemma 1. □

We now proceed to the properties concerning $\mathcal{U}_\mathcal{D}$ and $\mathcal{V}_\mathcal{D}$. For the Safety Theorem we need the following lemma, which is one of the two basic lemmas. The lemma shows how $\mathcal{U}_\mathcal{D}$ and $\mathcal{V}_\mathcal{D}$ approximate $\mathcal{T}_\mathcal{D}$.

LEMMA **4** $\forall t \in \mathcal{T}_\mathcal{D}[\![\ t_0\ ]\!]:\ N^0 \to \mathcal{B}[\![\ t\ ]\!] \in < \mathcal{U}_\mathcal{D}[\![\ N^0 \to \mathcal{B}[\![\ t_0\ ]\!]\ ]\!], \mathcal{V}_\mathcal{D}[\![\ N^0 \to \mathcal{B}[\![\ t_0\ ]\!]\ ]\!] >^{fo}$. □

PROOF: By cases on t_0. □

A few remarks seem appropriate.

First, it is really necessary to use the relation f. Consider what happens with $N^0 \to e[h\ t]$ when h is defined as $h\ v = g\ v$. Then $\mathcal{U}_\mathcal{D}[\![\ N^0 \to e[h\ t]\ ]\!] = \{N^0 \to e[N^h], N^h \to g\ N^v\}$ and $\mathcal{V}_\mathcal{D}[\![\ N^0 \to h\ t\ ]\!] = \{N^v \to t\}$. We cannot obtain the desired $N^0 \to g\ t$ by applying the c relation to the former and the d relation to the latter and combining with the s relation; the problem is that N^v occurs in a redex position. This is the phenomenon of the flow of data affecting the flow of control.

Second, the clause of $\mathcal{U}_\mathcal{D}$ for the case of an outermost constructor seems slightly inelegant. The problem is that the control grammar is approximating two things. First, via N^0 it approximates the set of terms encountered by $\mathcal{D}$. It is clear that if $\mathcal{D}$ encounters a term with an outermost constructor,

then in the next step it will encounter each of the arguments. Second, via N^h it approximates the result of unfolding h and proceeding with evaluation of the body. This terminates once a term with an outermost constructor has been reached, as previously explained, because the decomposition into context and redex then changes.

We now prove the Safety Theorem taking a certain lemma for granted. The lemma is subsequently proved; the proof of it elucidates the definition of $\diamond$.

THEOREM 7 *(Safety Theorem.)* $\forall t \in T^{\infty}_{\mathcal{D}} : \quad N^0 \to \mathcal{B}[\![\ t\ ]\!] \in < U^{\infty}_{\mathcal{D}}, V^{\infty}_{\mathcal{D}} >^o$. □

PROOF: First we prove by induction on i that

$$(*)\ \forall i\ \forall t \in T^i_{\mathcal{D}} : \quad N^0 \to \mathcal{B}[\![\ t\ ]\!] \in < U^i_{\mathcal{D}}, V^i_{\mathcal{D}} >^o$$

$i = 0$: obvious, since $N^0 \to \mathcal{B}[\![\ t_0\ ]\!] \in < \{N^0 \to \mathcal{B}[\![\ t_0\ ]\!]\}, \{\} >^o$.

$i = j + 1$: let $t_{j+1} \in T^{j+1}_{\mathcal{D}}$, *i.e.* $t_{j+1} \in \overline{\mathcal{T}}_{\mathcal{D}}(T^j_{\mathcal{D}})$. By the definition of $\mathcal{T}_{\mathcal{D}}$, we have $t_{j+1} \in \mathcal{T}_{\mathcal{D}}[\![\ t_j\ ]\!]$, for some $t_j \in T^j_{\mathcal{D}}$. By Lemma 4

$$N^0 \to \mathcal{B}[\![\ t_{j+1}\ ]\!] \in < \mathcal{U}_{\mathcal{D}}[\![\ N^0 \to \mathcal{B}[\![\ t_j\ ]\!]\ ]\!], \mathcal{V}_{\mathcal{D}}[\![\ N^0 \to \mathcal{B}[\![\ t_j\ ]\!]\ ]\!] >^{fo}$$

By the induction hypothesis, $N^0 \to \mathcal{B}[\![\ t_j\ ]\!] \in < U^j_{\mathcal{D}}, V^j_{\mathcal{D}} >^o$, so by the definition of $U_{\mathcal{D}}, V_{\mathcal{D}}$ and Lemma 1(1-4)

$$N \to \mathcal{B}[\![\ t_{j+1}\ ]\!] \in < \overline{\mathcal{U}}_{\mathcal{D}}(< U^j_{\mathcal{D}}, V^j_{\mathcal{D}} >^o), \overline{\mathcal{V}}_{\mathcal{D}}(< U^j_{\mathcal{D}}, V^j_{\mathcal{D}} >^o) >^{fo}$$

So by Lemma 6[1]

$$N^0 \to \mathcal{B}[\![\ t_{j+1}\ ]\!] \in < \overline{\mathcal{U}}_{\mathcal{D}}({U^j_{\mathcal{D}}}^{\diamond}) \cup U^j_{\mathcal{D}}, \overline{\mathcal{V}}_{\mathcal{D}}({U^j_{\mathcal{D}}}^{\diamond}) \cup V^j_{\mathcal{D}} >^{fo} \equiv < U^{j+1}_{\mathcal{D}}, V^{j+1}_{\mathcal{D}} >^o$$

Now we can prove the overall result using (*) as follows. If $t \in T^{\infty}_{\mathcal{D}}$, then $t \in T^i_{\mathcal{D}}$ for some i. Then, by (*), $N^0 \to \mathcal{B}[\![\ t\ ]\!] \in < U^i_{\mathcal{D}}, V^i_{\mathcal{D}} >^o$, and *a forteriori* by Lemma 1, $N^0 \to \mathcal{B}[\![\ t\ ]\!] \in < U^{\infty}_{\mathcal{D}}, V^{\infty}_{\mathcal{D}} >^o$. □

Notice the last step in the proof of (*). We might have defined our grammar construction as

$$< U^{j+1}_{\mathcal{D}}, V^{j+1}_{\mathcal{D}} > = < \overline{\mathcal{U}}_{\mathcal{D}}(< U^j_{\mathcal{D}}, V^j_{\mathcal{D}} >^o), \overline{\mathcal{V}}_{\mathcal{D}}(< U^j_{\mathcal{D}}, V^j_{\mathcal{D}} >^o) >^f$$

[1] This is the lemma we take for granted.

Then (*) pulls through trivially without the lemma. However, it may be that the computation of $< U_{\mathcal{D}}^j, V_{\mathcal{D}}^j >^o$ loops infinitely, preventing termination of our grammar construction. So we need to define $< U_{\mathcal{D}}^{j+1}, V_{\mathcal{D}}^{j+1} >$ in another manner. As can be seen from the last step of the proof of (*), all we require from $< U_{\mathcal{D}}^{j+1}, V_{\mathcal{D}}^{j+1} >$ is that

$$< \overline{\mathcal{U}}_{\mathcal{D}}(< U_{\mathcal{D}}^j, V_{\mathcal{D}}^j >)^o, \overline{\mathcal{V}}_{\mathcal{D}}(< U_{\mathcal{D}}^j, V_{\mathcal{D}}^j >^o) >^{fo} \subseteq < U_{\mathcal{D}}^{j+1}, V_{\mathcal{D}}^{j+1} >^o$$

So the idea is to get rid of as much as the computation of $< U_{\mathcal{D}}^j, V_{\mathcal{D}}^j >^o$ by replacing the o with a weaker relation utilizing the fact that the resulting pair of grammars is going to have fo applied to it.

The unproved result refered to states that we can choose $< U_{\mathcal{D}}^{j+1}, V_{\mathcal{D}}^{j+1} >$ as

$$< \overline{\mathcal{U}}_{\mathcal{D}}(U_{\mathcal{D}}^{j\,\diamond}) \cup U_{\mathcal{D}}^j \,,\, \overline{\mathcal{V}}_{\mathcal{D}}(U_{\mathcal{D}}^{j\,\diamond}) \cup V_{\mathcal{D}}^j >^f$$

ie that

$$< \overline{\mathcal{U}}_{\mathcal{D}}(< U_{\mathcal{D}}^j, V_{\mathcal{D}}^j >^o), \overline{\mathcal{V}}_{\mathcal{D}}(< U_{\mathcal{D}}^j, V_{\mathcal{D}}^j >^o) >^{fo} \subseteq < \overline{\mathcal{U}}_{\mathcal{D}}(U_{\mathcal{D}}^{j\,\diamond}) \cup U_{\mathcal{D}}^j, \overline{\mathcal{V}}_{\mathcal{D}}(U_{\mathcal{D}}^{j\,\diamond}) \cup V_{\mathcal{D}}^j >^{fo}$$

To prove that result, we need the following lemma, which is the other basic lemma.

LEMMA **5**

$$\begin{array}{lcll}
\mathcal{U}_{\mathcal{D}}[\![\, M \to e_1[e_2[r]] \,]\!] & \subseteq & (\mathcal{U}_{\mathcal{D}}[\![\, N \to e_2[r] \,]\!] \cup \{M \to e_1[N]\})^c & (1) \\
\mathcal{U}_{\mathcal{D}}[\![\, M \to e_1[e_2[o]] \,]\!] & \subseteq & (\mathcal{U}_{\mathcal{D}}[\![\, N \to e_2[o] \,]\!] \cup \{M \to e_1[N]\})^c, \ \textit{if } e_2 \neq [] & (4) \\
\mathcal{U}_{\mathcal{D}}[\![\, M \to e(\overline{t})[r(\overline{s})] \,]\!] & \subseteq & < \mathcal{U}_{\mathcal{D}}[\![\, M \to e(\overline{N})[r(\overline{K})] \,]\!], \overline{N} \to \overline{t} \cup \overline{K} \to \overline{s} >^{fs} & (2) \\
\mathcal{U}_{\mathcal{D}}[\![\, M \to e(\overline{t})[o(\overline{s})] \,]\!] & \subseteq & < \mathcal{U}_{\mathcal{D}}[\![\, M \to e(\overline{N})[o(\overline{K})] \,]\!], \overline{N} \to \overline{t} \cup \overline{K} \to \overline{s} >^{fs}, \ \textit{if } e \neq [] & (5) \\
\mathcal{U}_{\mathcal{D}}[\![\, M \to o(\overline{t}) \,]\!] & \subseteq & < \mathcal{U}_{\mathcal{D}}[\![\, M \to o(\overline{N}) \,]\!], \overline{N} \to \overline{t} >^{fs} \ \textit{if } M \equiv N^0 & (3a) \\
 & \subseteq & \equiv \{\} \ \textit{otherwise} & (3b)
\end{array}$$

$$\begin{array}{lcll}
\mathcal{V}_{\mathcal{D}}[\![\, M \to e_1[e_2[r]] \,]\!] & \subseteq & \mathcal{V}_{\mathcal{D}}[\![\, N \to e_2[r] \,]\!] & (6) \\
\mathcal{V}_{\mathcal{D}}[\![\, M \to e_1[e_2[o]] \,]\!] & \subseteq & \mathcal{V}_{\mathcal{D}}[\![\, N \to e_2[o] \,]\!], \ \textit{if } e_2 \neq [] & (9) \\
\mathcal{V}_{\mathcal{D}}[\![\, M \to e(\overline{t})[r(\overline{s})] \,]\!] & \subseteq & (\mathcal{V}_{\mathcal{D}}[\![\, M \to e(\overline{N})[r(\overline{K})] \,]\!] \cup \overline{K} \to \overline{s})^d & (7) \\
\mathcal{V}_{\mathcal{D}}[\![\, M \to e(\overline{t})[o(\overline{s})] \,]\!] & \subseteq & (\mathcal{V}_{\mathcal{D}}[\![\, M \to e(\overline{N})[o(\overline{K})] \,]\!] \cup \overline{N} \to \overline{t} \cup \overline{K} \to \overline{s})^d, \ \textit{if } e \neq [] & (10) \\
\mathcal{V}_{\mathcal{D}}[\![\, M \to o(\overline{t}) \,]\!] & = & \{\} & (8)
\end{array}$$

□

PROOF: (1), (6): By cases on r. (2), (7): By cases on r. (4), (9): By (1), (6), since $e_2[o]$ can be parsed into $e_2'[r]$. (5), (10): By (2), (7). (3), (8): By cases on o. □

This lemma allows us to get rid of derivations. Consider, for instance, inclusion (1). In each step of the grammar construction we extend the control grammar by performing certain derivations, before calculating the new productions with the $\mathcal{U}_\mathcal{D}$ and $\mathcal{V}_\mathcal{D}$ functions. (1) shows that it is not necessary to derive the production $M \to e_1[e_2[r]]$ from productions $N \to e_2[r], M \to e_1[N]$ because all we get from applying $\mathcal{U}_\mathcal{D}$ to the former production we also get from applying $\mathcal{U}_\mathcal{D}$ to $N \to e_2[r]$ as long as we retain all the original productions (specifically $M \to e_1[N]$) and use the c relation after applying $\mathcal{U}_\mathcal{D}$.

This is systematized in the next lemma. The $\diamond$ relation may be thought of as the relation that computes everything we couldn't get rid of in the proof of the next lemma.

However, before proceeding a few remarks on the o relation may turn out helpful. Recall that o is factored into c, d and s. *First*, o makes a derivation in C^c. This will always yield a production of form:

$$M_0 \to e_1(\overline{N}_1)[e_2(\overline{N}_2)[\ldots e_k(\overline{N}_k)[e_{k+1}(\overline{N}_{k+1})]\ldots]]$$

where for $i = 0, \ldots k-1, M_i \to e_{i+1}(\overline{N}_{i+1})[M_{i+1}] \in C$, $M_k \to e_{k+1}(\overline{N}_{k+1}) \in C$. *Second*, o makes a number of derivations $\overline{N}_i \to \overline{t}_i \in D^d$. *Finally,* it combines the latter with the former in one step yielding a production of form:

$$M' \to e_1(\overline{t}_1)[e_2(\overline{t}_2)[\ldots e_k(\overline{t}_k)[e_{k+1}(\overline{t}_{k+1})]\ldots]]$$

Diagrammatically:

$$\begin{array}{ccccccc} M_0 \overset{C}{\to} & e_1(\overline{N}_1)[M_1] & M_1 \overset{C}{\to} & e_2(\overline{N}_2)[M_2] & \ldots & M_k \overset{C}{\to} & e_{k+1}(\overline{N}_{k+1}) \\ & \downarrow & & \downarrow & & & \downarrow \\ & \overline{N}_1 \overset{D^d}{\to} \overline{t}_1 & & \overline{N}_2 \overset{D^d}{\to} \overline{t}_2 & & & \overline{N}_{k+1} \overset{D^d}{\to} \overline{t}_{k+1} \end{array}$$

The term $e_{k+1}(\overline{N}_{k+1})$ is called the *critical term* of the derivation in o.

Note that in the combining step, the derivations from D^d are not allowed to be used in the strict position of the grammar term from C^c. Also, note that the decomposition of the term $e'[e(\overline{t})[r(\overline{s})]]$ into context and redex gives $e'[e(\overline{t})[]]$ as context and $r(\overline{s})$ as redex.

LEMMA 6 $< \overline{\mathcal{U}}_{\mathcal{D}}(< C, D >^o), \overline{\mathcal{V}}_{\mathcal{D}}(< C, D >^o) >^{fo} \subseteq < \overline{\mathcal{U}}_{\mathcal{D}}(C^\diamond) \cup C, \overline{\mathcal{V}}_{\mathcal{D}}(C^\diamond) \cup D >^{fo}$ $\Box$

PROOF: By Corollary 4 and Lemma 1 it suffices to show that

$$\begin{array}{lll} \overline{\mathcal{U}}_{\mathcal{D}}(< C, D >^o) & \subseteq & < (\overline{\mathcal{U}}_{\mathcal{D}}(C^\diamond) \cup C)^c, D^d >^{fs} \\ \overline{\mathcal{V}}_{\mathcal{D}}(< C, D >^o) & \subseteq & (\overline{\mathcal{V}}_{\mathcal{D}}(C^\diamond) \cup D)^d \end{array}$$

Using the definition of $\overline{\mathcal{U}}, \overline{\mathcal{V}}$ it is easy to see that this will follow, if we prove:

$\forall M_0 \to t' \in < C, D >^o \ \exists M_0' \to t'' \in C^\diamond$:

$$\begin{array}{lll} \mathcal{U}_{\mathcal{D}}[\![M_0 \to t']\!] & \subseteq & < (\mathcal{U}_{\mathcal{D}}[\![M_0' \to t'']\!] \cup C)^c, D^d >^{fs} \\ \mathcal{V}_{\mathcal{D}}[\![M_0 \to t']\!] & \subseteq & (\mathcal{V}_{\mathcal{D}}[\![M_0' \to t'']\!] \cup D)^d \end{array}$$

We prove this by cases on the critical term of t'.

Case 1: context-nonterminal term: $(k \geq 0)$

$$\begin{array}{ccccc} M_0 \to & e_1(\overline{N}_1)[M_1] & \ldots & M_k \to & e_{k+1}(\overline{N}_{k+1})[M_{k+1}] \\ & \downarrow & & & \downarrow \\ & \overline{N}_1 \to \overline{t}_1 & & & \overline{N}_{k+1} \to \overline{t}_{k+1} \end{array}$$

Then $t' \equiv e_1(\overline{t}_1)[\ldots[e_k(\overline{t}_{k+1})[M_{k+1}]]\ldots]$, and

$$\begin{array}{lll} \mathcal{U}_{\mathcal{D}}[\![M_0 \to e_1(\overline{t}_1)[\ldots[e_k(\overline{t}_{k+1})[M_{k+1}]]\ldots]]\!] & = & \{\} \\ \mathcal{V}_{\mathcal{D}}[\![M_0 \to e_1(\overline{t}_1)[\ldots[e_k(\overline{t}_{k+1})[M_{k+1}]]\ldots]]\!] & = & \{\} \end{array}$$

So any element of $C^\diamond$ will do the job. Notice that the case $k = 0$ is included.

Case 2: context-redex term: $(k \geq 0)$

$$\begin{array}{ccccc} M_0 \to & e_1(\overline{N}_1)[M_1] & \ldots & M_k \to & e_{k+1}[r(\overline{K})] \\ & \downarrow & & & \downarrow \quad \downarrow \\ & \overline{N}_1 \to \overline{t}_1 & & & \overline{N}_{k+1} \to \overline{t}_{k+1} \ \overline{K} \to \overline{s} \end{array}$$

Then $t' \equiv e_1(\overline{t}_1)[\ldots[e_{k+1}(\overline{t}_{k+1})[r(\overline{s})]]\ldots]$, and

$\mathcal{U}_{\mathcal{D}}[\![M_0 \to e_1(\overline{t}_1)[\ldots[e_{k+1}(\overline{t}_{k+1})[r(\overline{s})]]\ldots]]\!] \subseteq$
$< (\mathcal{U}_{\mathcal{D}}[\![M_k \to e_{k+1}(\overline{N_{k+1}})[r(\overline{K})]]\!] \cup \{M_0 \to e_1(\overline{N}_1)[\ldots[e_k(\overline{N}_k)[\overline{M}_k]]\ldots]\})^c,$
$\cup_{i=1}^{k+1}\{\overline{N_i} \to \overline{t_i}\} \cup \overline{K} \to \overline{s} >^{fs}$

$\mathcal{V}_{\mathcal{D}}[\![M_0 \to e_1(\overline{t}_1)[\ldots[e_{k+1}(\overline{t}_{k+1})[r(\overline{s})]]\ldots]]\!] \subseteq$
$(\mathcal{V}_{\mathcal{D}}[\![M_k \to e_{k+1}(\overline{N_{k+1}})[r(\overline{K})]]\!] \cup \overline{K} \to \overline{s})^d$

by Lemma 5(1-2),(6-7). Lemma 1 and Lemma 2(4-5) now yield the desired.

Case 3: observable term: $(k \geq 0)$

$$\begin{array}{ccccccccc} M_0 \to & e_1(\overline{N}_1)[M_1] & \ldots & M_{k-1} \to & e_k(\overline{N}_k)[M_k] & M_k \to & o(\overline{K}) \\ & \downarrow & & & \downarrow & & \downarrow \\ & \overline{N}_1 \to \overline{t}_1 & & & \overline{N}_k \to \overline{t}_k & & \overline{K} \to \overline{s} \end{array}$$

Here we consider three subcases.

Subcase A: $e_1 = \ldots = e_k = []$, $M_0 \equiv N^0$. Then $t' \equiv o(\overline{s})$, and

$$\begin{array}{rcl} \mathcal{U}_{\mathcal{D}}[\![\ M_0 \to o(\overline{s})\ ]\!] & \subseteq & < E_C[\![\ M_0 \to o(\overline{K})\ ]\!], \overline{K} \to \overline{s} >^{fs} \\ \mathcal{V}_{\mathcal{D}}[\![\ M_0 \to o(\overline{s})\ ]\!] & \equiv & \{\} \end{array}$$

by Lemma 5(3),(8). Lemma 1 and Lemma 2(4-5) now yield the desired.

Subcase B: $e_1 = \ldots = e_k = []$, $M_0 \not\equiv N^0$. Then $t' \equiv o(\overline{s})$, and

$$\begin{array}{rcl} \mathcal{U}_{\mathcal{D}}[\![\ M_0 \to o(\overline{s})\ ]\!] & \equiv & \{\} \\ \mathcal{V}_{\mathcal{D}}[\![\ M_0 \to o(\overline{s})\ ]\!] & \equiv & \{\} \end{array}$$

by Lemma 5(3),(8), so any element of $C^{\diamond}$ will do.

Subcase C: $e_k \neq []$ for some $i \in \{1 \ldots k\}$. Let l be the largest number with this property. Then $t' \equiv e_1(\overline{t}_1)[\ldots[e_l(\overline{t}_l)[o(\overline{s})]]\ldots]$, and

$$\begin{array}{l} \mathcal{U}_{\mathcal{D}}[\![\ M_0 \to e_1(\overline{t}_1)[\ldots[e_l(\overline{t}_l)[o(\overline{s})]]\ldots]\ ]\!] \subseteq \\ < (\mathcal{U}_{\mathcal{D}}[\![\ M_{l-1} \to e_l(\overline{N}_l)[o(\overline{K})]\ ]\!] \cup \{M_0 \to e_1(\overline{t}_1)[\ldots[e_{l-1}(\overline{t}_{l-1})[\overline{M}_{l-1}]]\ldots]\})^c, \\ \overline{K} \to \overline{s} \cup_{i=1}^{l} \overline{N}_i \to \overline{t}_i >^{fs} \end{array}$$

$$\begin{array}{l} \mathcal{V}_{\mathcal{D}}[\![\ M_0 \to e_1(\overline{t}_1)[\ldots[e_l(\overline{t}_l)[o(\overline{s})]]\ldots]\ ]\!] \subseteq \\ (\mathcal{V}_{\mathcal{D}}[\![\ M_{l-1} \to e_l(\overline{N}_l)[o(\overline{K})]\ ]\!] \cup \overline{K} \to \overline{s} \cup_{i=1}^{l} \overline{N}_i \to \overline{t}_i)^d \end{array}$$

by Lemma 5(4-5),(9-10). Lemma 1 and Lemma 2(4-5) now yield the desired. □

This was the lemma we needed to prove; we have thereby concluded the proof of safety of the approximation for $\mathcal{D}$.

THEOREM 8 *(Safety Theorem.)* $\forall t \in T_{\mathcal{H}}^{\infty} : N^0 \to \mathcal{B}[\![\ t\ ]\!] \in < U_{\mathcal{H}}^{\infty}, V_{\mathcal{H}}^{\infty} >^o$. □

PROOF: Note that the only properties about $\mathcal{U}_\mathcal{D}$ and $\mathcal{V}_\mathcal{D}$ used in the proof of Lemma 6 were those from Lemma 5. Recall that the only properties about $\mathcal{U}$ and $\mathcal{V}$ used in Theorem 7 were those from Lemma 4 and those from Lemma 6. Those from Lemma 6, in turn, all stemmed from Lemma 5, so to reprove Theorem 7 for $\mathcal{H}$ we need only reprove Lemma 4 and Lemma 5, which we called the basic lemmas. Generalized versions of the basic lemmas are easy to prove using the basic lemmas themselves. □

8.2 Termination

The purpose of this section is to prove that the algorithm computing the grammar which approximates $T^\infty_\mathcal{H}$ always terminates with a finite grammar. More precisely, we prove that

$$\exists j : < \cup^j_{i=0} U^i_\mathcal{H}, \cup^j_{i=0} V^i_\mathcal{H} > = < \cup^\infty_{i=0} U^i_\mathcal{H}, \cup^\infty_{i=0} V^i_\mathcal{H} >$$

First we define a measure on right hand sides of productions.

DEFINITION **25** (Size of grammar terms.)

$$\begin{array}{lcl}
\mathcal{S}[\![\bullet]\!] & = & 1 \\
\mathcal{S}[\![c\, t_1 \ldots t_n]\!] & = & 1 + \sum^n_{i=1} \mathcal{S}[\![t_i]\!] \\
\mathcal{S}[\![f\, t_1 \ldots t_n]\!] & = & 1 + \sum^n_{i=1} \mathcal{S}[\![t_i]\!] \\
\mathcal{S}[\![g\, t_0 \ldots t_n]\!] & = & 1 + \sum^n_{i=0} \mathcal{S}[\![t_i]\!] \\
\mathcal{S}[\![N]\!] & = & 1
\end{array}$$

We use $n_1 \vee n_2 \ldots \vee n_k$ to denote the maximum of the numbers $n_1, n_2 \ldots n_k$.

$$\overline{\mathcal{S}}(Q) \;=\; \vee_{N \to t \in Q} \mathcal{S}[\![t]\!]$$

In the context of some term t and program p with right hand sides $t_1 \ldots t_n$ we let $\mathcal{S}_0$ denote $\mathcal{S}[\![\mathcal{B}[\![t]\!]]\!] \vee \mathcal{S}[\![\mathcal{B}[\![t_1]\!]]\!] \vee \ldots \vee \mathcal{S}[\![\mathcal{B}[\![t_n]\!]]\!]$, *i.e.* $\mathcal{S}_0$ denotes the size of the largest term among t and the right hand sides of the program. □

As usual we need some lemmas to do the dirty work.

LEMMA 7

$$\begin{array}{lll} \textit{(1)} & \overline{\mathcal{S}}(\mathcal{V}_{\mathcal{H}}[\![\ N \to t\ ]\!]) & \leq \ \mathcal{S}[\![\ t\ ]\!] \\ \textit{(2)} & \overline{\mathcal{S}}(\mathcal{U}_{\mathcal{H}}[\![\ N \to t\ ]\!]) & \leq \ \mathcal{S}[\![\ t\ ]\!] \vee \mathcal{S}_0 \\ & & \\ \textit{(3)} & \overline{\mathcal{S}}(\mathcal{V}_{\mathcal{H}}[\![\ N \to e[s]\ ]\!]) & \leq \ \mathcal{S}[\![\ e[N]\ ]\!] \vee \mathcal{S}[\![\ s\ ]\!] \\ \textit{(4)} & \overline{\mathcal{S}}(\mathcal{U}_{\mathcal{H}}[\![\ N \to e[s]\ ]\!]) & \leq \ \mathcal{S}[\![\ e[N]\ ]\!] \vee \mathcal{S}[\![\ s\ ]\!] \vee \mathcal{S}_0 \end{array}$$

where in (3),(4) s is one of $\bullet, N', c\,t_1 \ldots t_n$ and N is an arbitrary nonterminal.
□

PROOF: (1),(2): First prove the assertion for $\mathcal{V}_{\mathcal{D}}, \mathcal{U}_{\mathcal{D}}$ by cases on t, then extend it to $\mathcal{V}_{\mathcal{H}}, \mathcal{U}_{\mathcal{H}}$. (3),(4): First prove the assertion for $\mathcal{V}_{\mathcal{D}}, \mathcal{U}_{\mathcal{D}}$ by cases on s, then extend it to $\mathcal{V}_{\mathcal{H}}, \mathcal{U}_{\mathcal{H}}$. □

LEMMA 8

$$\begin{array}{lll} \overline{\mathcal{S}}(\overline{\mathcal{V}}_{\mathcal{H}}(C^{\diamond})) & \leq & \overline{\mathcal{S}}(C) \\ \overline{\mathcal{S}}(\overline{\mathcal{U}}_{\mathcal{H}}(C^{\diamond})) & \leq & \overline{\mathcal{S}}(C) \vee \mathcal{S}_0 \end{array}$$

□

PROOF: We must show that

$$\forall N \to t \in C^{\diamond}\ \exists N' \to t' \in C :\ \overline{\mathcal{S}}(\mathcal{V}_{\mathcal{H}}[\![\ N \to t\ ]\!]) \leq \mathcal{S}[\![\ t'\ ]\!]$$

$$\forall N \to t \in C^{\diamond}\ \exists N' \to t' \in C :\ \overline{\mathcal{S}}(\mathcal{U}_{\mathcal{H}}[\![\ N \to t\ ]\!]) \leq \mathcal{S}[\![\ t'\ ]\!] \vee \mathcal{S}_0$$

First note that if $N \to t \in C^{\diamond}$ then either $N \to t \in C$, or $t \equiv e[s]$ where $N \to e[M] \in C$ and $M' \to s \in C$ and s is one of $\bullet, M'', c\,t_1 \ldots t_n$. In the former case use Lemma 7(1-2), in the latter Lemma 7(3-4). □

LEMMA 9

$$\begin{array}{lll} \overline{\mathcal{S}}(\cup_{i=0}^{\infty} U_{\mathcal{H}}^i) & \leq & \mathcal{S}_0 \\ \overline{\mathcal{S}}(\cup_{i=0}^{\infty} V_{\mathcal{H}}^i) & \leq & \mathcal{S}_0 \end{array}$$

□

PROOF: First use Lemma 8 to prove by induction on j

$$\begin{array}{lll} (1) \quad \forall j: \ \overline{\mathcal{S}}(\cup_{i=0}^{j} U_{\mathcal{H}}^{i}) & \leq & \mathcal{S}_0 \\ (2) \quad \forall j: \ \overline{\mathcal{S}}(\cup_{i=0}^{j} V_{\mathcal{H}}^{i}) & \leq & \mathcal{S}_0 \end{array}$$

To prove the first part of the overall assertion, let $N \rightarrow t \in \cup_{i=0}^{\infty} V_{\mathcal{H}}^{i}$. Then $N \rightarrow t \in \cup_{i=0}^{j} V_{\mathcal{H}}^{i}$ for some j. Then by (1) $\mathcal{S}[\![t]\!] \leq \mathcal{S}_0$, as desired. The second part follows similarly. □

REMARK **6** Note in the preceding proof that we find an upper bound on the size of any term in the possibly infinite grammars $\cup_{i=0}^{j} U_{\mathcal{H}}^{i}$ and $\cup_{i=0}^{j} V_{\mathcal{H}}^{i}$ by finding an upper bound on the size of any term in the finite subgrammars. This only works because the upper bound is the same, viz $\mathcal{S}_0$, in all the finite subgrammars.

We can express this by saying that the property of a grammar that the size of any right hand side be bounded by a certain number n, has finite character. In the preceding section a similar correspondence occurred in the Safety Theorem, where we deduced

$$\forall t \in T_{\mathcal{D}}^{\infty}: \quad N^0 \rightarrow \mathcal{B}[\![t]\!] \in < U_{\mathcal{D}}^{\infty}, V_{\mathcal{D}}^{\infty} >^{o}$$

from

$$\forall i \, \forall t \in T_{\mathcal{D}}^{i}: \quad N^0 \rightarrow \mathcal{B}[\![t]\!] \in < U_{\mathcal{D}}^{i}, V_{\mathcal{D}}^{i} >^{o}$$

□

THEOREM **9**

$$\exists j : < \cup_{i=0}^{j} U_{\mathcal{H}}^{i}, \cup_{i=0}^{j} V_{\mathcal{H}}^{i} > = < \cup_{i=0}^{\infty} U_{\mathcal{H}}^{i}, \cup_{i=0}^{\infty} V_{\mathcal{H}}^{i} >$$

□

PROOF: Note that the right hand sides of the computed grammars use only the following symbols: function and constructor symbols from the given term and program; the $\bullet$ symbol; the nonterminals N^v and N^h where v and h are variable and function symbols, respectively, from the given term and program. Since there are only finitely many of these symbols, only a finite number of right hand sides with size bounded by $\mathcal{S}_0$ are possible.

So the right hand side of the equation contains only finitely many productions. Obviously, these are added in a finite number of steps. □

Note that it is critical in the preceding proof that the right hand sides of the productions are constructed using a finite alphabet of symbols. This would not be the case if we had used variable names instead of •, because the extraction mechanism invents new variable names.

Chapter 9

Improvements of our method

This chapter describes a number of natural improvements of the method described in Chapter 6. There are two essentially different ways of improving the method: improving the accuracy of the computed grammar, and improving the technique for finding annotations. Of course, improving the accuracy will lead to better annotations and thereby more transformed programs.

The first and last section describes improvements of the former kind; the second section describes an improvement of the latter kind.

Apart from improvement of the output of the deforestation algorithm, these extensions are motivated by the desire to make our method at least as good as certain previous methods. We amplify this point in the next chapter.

9.1 Tracing the top-level arguments

Consider the following term and program.

$$
\begin{array}{lcl}
 & & f\,(f'\,z) \\
f'\,w & = & f\,w \\
f\,v & = & g\,v \\
g\,(C\,u) & = & u
\end{array}
$$

The transformation proceeds as follows.

$$
\begin{array}{lcl}
\mathcal{G}[\![\, f\,(f'\,z)\,]\!] & \Rightarrow & \mathcal{G}[\![\, g\,(f'\,z)\,]\!] \\
 & \Rightarrow & \mathcal{G}[\![\, g\,(f\,z)\,]\!] \\
 & \Rightarrow & \mathcal{G}[\![\, g\,(g\,z)\,]\!] \\
 & \Rightarrow & g'\,z \\
\text{where} & & \\
g'\,(C\,u) & = & \mathcal{G}[\![\, g\,u\,]\!] \\
 & = & g''\,u \\
\quad \text{where} & & \\
\quad g''\,(C\,u) & = & u
\end{array}
$$

It is evident that there is no termination problem in transforming this term and program.

Recall that the grammar construction does not keep track of when the different variable bindings are made, or where function calls come from. Therefore, it looks to the grammar construction like the evaluation of the call $f\ldots$ to f with some argument has led to a term $g\,(f\ \ldots)$ with a call to f in the redex after the second step. In fact the control grammar, C, computed from this term and program is:

$$
\begin{array}{lcl}
N^0 & \rightarrow & f\,(f'\,\bullet) \mid N^f \\
N^{f'} & \rightarrow & f\,N^w \mid N^f \\
N^f & \rightarrow & g\,N^v \mid N^g \\
N^g & \rightarrow & \bullet \\
N^v & \rightarrow & f'\,\bullet \mid N^{f'} \mid N^w \\
N^w & \rightarrow & \bullet
\end{array}
$$

with $N^f \rightarrow g\,N^f \in C^c$ signifying that f should be extracted.

But we saw above that there was no problem. We have cheated the grammar construction by having an initial term ft which unfolds to $e[t]$ where t reduces to a second call to f; the second call to f is not "caused" in any way by the first. So the problem is to make sure that when $N^f \rightarrow g\,N^f \in C^c$, then the second call really is caused by the first in some way.

The solution is theoretically very simple.[1] Suppose that we wish to transform the term t in program p and that t contains, syntactically, n function

[1] An actual implementation should be more clever on certain points; we return to this after describing the extension.

calls. Take n copies of p. In the i'th copy, all function names and variable names are subscripted with i. In the term, subscript each function call with a unique number i between 1 and n, informally signifying that this call be evaluated in the i'th copy of p. Then compute the grammar. Now look for infinity in this grammar. For instance, $N^{f_1} \to e[N^{f_2}]$ does not signify infinity but $N^{f_1} \to e[N^{f_1}]$ does. Put annotations on the n copies of the program accordingly, and transform the term using the n copies of the program.

Note that there is nothing new in this method, theoretically. Before transformation, we simply make n copies of the program, and change the orginal calls in the term into calls to the different versions.

For the example, transformation with no annotations proceeds as follows:

$$\begin{array}{lcl}
\mathcal{G}[\![\ f_1\ (f_2'\ z)\ ]\!] & \Rightarrow & \mathcal{G}[\![\ g_1\ (f_2'\ z)\ ]\!] \\
 & \Rightarrow & \mathcal{G}[\![\ g_1\ (f_2\ z)\ ]\!] \\
 & \Rightarrow & \mathcal{G}[\![\ g_1\ (g_2\ z)\ ]\!] \\
 & \Rightarrow & g'\ z \\
\text{where} & & \\
g'\ (C\ u) & = & \mathcal{G}[\![\ g_1\ u\ ]\!] \\
 & = & g''\ u \\
\quad \text{where} & & \\
\quad g''\ (C\ u) & = & u
\end{array}$$

Here we see the important point: now the two calls to f are separable—they have different numbers. Indeed the new control grammar is:

$$\begin{array}{lcl}
N^0 & \to & f_1\ (f_2'\ \bullet) \mid N^{f_1} \\
N^{f_2'} & \to & f_2\ N^{w_2} \mid N^{f_2} \\
N^{f_1} & \to & g_1\ N^{v_1} \mid N^{g_1} \\
N^{f_2} & \to & g_2\ N^{v_2} \mid N^{g_2} \\
N^{g_1} & \to & \bullet \\
N^{g_2} & \to & \bullet \\
N^{v_1} & \to & f_2'\ \bullet \mid N^{f_2'} \\
N^{v_2} & \to & N^{w_2} \\
N^{w_2} & \to & \bullet
\end{array}$$

As expected, the grammar does not contain infinity.

In the above example, no nested function calls were present in the right hand sides of the definitions of the program. However, suppose that we

introduced a new function $h\ x = f\ (f'\ x)$ and ask that the term $h\ x$ be transformed. Now we have the same problem as before, but we cannot use the same idea as before to solve the problem. Suppose that n and m are the number of function calls in the term and program, respectively. If we were to use the same idea we would first have to make n copies as before. For each of these n copies we would have to make m copies, and for each of these $n \times m$ copies we would have to make m copies, *etc.* Informally, each time a set of copies is made, the problem is transfered one level down in the calling hierachy.

However, the case considered above where we make only n copies is particularly interesting, because in Section 10.6 we prove that with this extension our method is strictly better than Wadler's orginal means of ensuring termination; *without* the extension our method is *not* strictly better. Apart from that result we shall not attempt to prove any general theorem concerning the quality of the extended method.

Let us briefly see how we can avoid making n copies of the entire program.

First, the grammar algorithm need not work with n copies of the program. The function names in the initial term *should* have numbers, but the rules computing new productions should simply propagate these numbers to the productions in a suitable manner.

Second, there is the question whether the productions occurring several times should really be represented several times. An alternative is to attach to each production a suitable set of integer tuples encoding the versions of the different nonterminals for which the production is present in the grammar.

The result of this grammar is still annotations for n programs. We may *e.g.* have $N^{f_1} \to e[N^{f_1}] \in C^C$ and not $N^{f_2} \to e[N^{f_2}] \in C^C$ at the same time. It is clear that if the transformation using n different annotated versions of the program terminates, then so does the transformation using only one version with all the annotations united;[2] the difference is that in the latter case *less* unfolding and binding is done because of the united annotations, and *more* folding is done, because there is no distinction between different versions of a function.

This concludes the first extension which we call *tracing the top-level* arguments.

[2]That is, f may only be unfolded if it may be unfolded in all the n annotated programs.

9.2 Breaking cycles with care

When there is infinity in the control grammar, *i.e.* $N^f \to e[N^f] \in C^c$, it stems from a derivation $N^f \to e_1[N^1], N^1 \to e_2[N^2] \ldots N_k \to e_k[N^f]$. But then it will also be the case that $N^h \to e'[N^h]$ for all the other nonterminals for functions among $N^1 \ldots N^k$. The means of preventing infinity used in Theorem 5 is currently to prevent unfolding of *all* these functions. However, less may do the job. We may select just *one* of the nonterminals for functions and then recalculate the grammar, instead of selecting all the nonterminals and then recalculating the grammar. This may lead to more recalculation.

The correctness of this method follows directly from the correctness of the previous method, since we still keep annotating the program until there is no infinity in the resulting program.

Which nonterminal should be chosen above? Different strategies are possible. For instance, we might always pick a nonterminal N^f for which there is a call to f in the program with a non-variable argument. Another idea is to pick a nonterminal N^f which is currently lying in the largest number of cycles $N^f \to e_1[N^1], N^1 \to e_2[N^2] \ldots N_k \to e_k[N^f]$. Yet other strategies are possible.

This extension is required if our method is to find annotations no worse than those which Chin's Polyvariant annotation scheme (see the next chapter) finds. This concludes the second extension which we call *breaking cycles with care.*

9.3 Using polyvariant annotations

Recall that in Section 7.6 we looked for one annotation (one set of parameters to be extracted and one $\oplus/\ominus$) for each function *definition* in the program. Another idea is to look for one annotation for each function *call* in the program. The former annotation is, in the terminology of partial evaluation, *monovariant*, while the latter is *polyvariant.*

The grammar construction is changed as follows. Instead of a nonterminal N^f for each function definition and a nonterminal N^v for each formal parameter of a function definition, there is a nonterminal N^{f_i} for each call to f in the program and term and corresponding N^{v_i}. When adding new productions for eg N^f, the grammar construction must know which call to f

was unfolded. This can be achieved simply by numbering the calls according to the numbering of the nonterminals.

It does not seem hard to change the correctness proof of the previous monovariant method into a correctness proof of this extension which we call *using polyvariant annotations.*

This extension is required if our method is to find annotations no worse than those which Chin's Polyvariant annotation scheme (see the next chapter) finds. This concludes the second extension which we call *breaking cycles with care.*

Chapter 10

Previous means of ensuring termination

This chapter describes previous means of ensuring termination of the deforestation algorithm. The last section relates our method to (some of) the previous methods.

There is a tradition in research articles which would have it that the section or chapter concerning previous methods be of a somewhat handwaving character. Since, however, we attempt in the last section of this chapter to relate our method to previous methods quite precisely, this chapter gives correspondingly precise descriptions of the previous methods. Any errors or misperceptions in these descriptions are of course entirely the responsibility of the author of the present paper.

10.1 Wadler's treeless terms

This section describes the solution to the termination problem proposed in [Fer88]. The solution consists of restricting application of $\mathcal{F}$ to a certain class of programs for which termination can be guaranteed.

DEFINITION **26** (Treelessness.) Define a program p to be *treeless* [*restricted treeless*] if every right hand side of every definition of p conforms to the

grammar of tt [rtt] below:

$$\begin{array}{lcl} tt & ::= & v \mid c\ tt_1 \ldots tt_n \mid f\ v_1 \ldots v_n \mid g\ v_0 \ldots v_n \\ rtt & ::= & rtt' \mid c\ rtt'_1 \ldots rtt'_n \\ rtt' & ::= & v \mid f\ v_1 \ldots v_n \mid g\ v_0 \ldots v_n \end{array}$$

where each function called in tt [rtt] is defined in p.[1] Each right hand side of p is called a treeless [restricted treeless] term if p is treeless [restricted treeless]. □

Note that in a treeless term all function calls have only variables as arguments. A restricted treeless term has the further restriction that it contains at most one constructor which, if present, occurs outermost. The requirement of having only variable arguments means that the program does not build any intermediate structures, since this would require the presence of a subterm $f\ (f'x)$ with f' constructing and f destructing. The other requirement means that terms won't grow bigger during transformation (this will be made precise below.)

DEFINITION **27** Define function terms, ft:

$$ft \ ::= \ v \mid f\ ft_1 \ldots ft_n \mid g\ ft_0 \ldots ft_n$$

□

The major result in [Fer88] is:

THEOREM **10** *(Deforestation Theorem.) For a function term ft and a restricted treeless program p $\mathcal{F}[\![\ ft\ ]\!]$ terminates.* □

REMARK **7** [Fer88] also asserted that under the conditions of the preceding theorem: (1) the result of the transformation is a treeless term and new treeless functions; (2) if, in addition, every function in p is linear, the resulting term is at least as efficient as the original ft. The former can be proved by induction the argument term proceeding by cases. The latter was not proved in [Fer88], and will not be proved here; some consideration of the problem was given in Chapter 5. □

[1] The reader familiar with [Fer88] may note that we do not take the linearity requirement as a part of the definition of treelessness.

The proof uses two lemmas which we first state.

DEFINITION **28** Define the *nesting* of a term t, $\mathcal{N}[\![\ t\ ]\!]$, by

$$\begin{array}{lcl}
\mathcal{N}[\![\ v\ ]\!] & = & 0 \\
\mathcal{N}[\![\ c\, t_1 \ldots t_n\ ]\!] & = & \max\{\mathcal{N}[\![\ t_1\ ]\!] \ldots \mathcal{N}[\![\ t_n\ ]\!]\} \\
\mathcal{N}[\![\ f\, t_1 \ldots t_n\ ]\!] & = & 1 + \max\{\mathcal{N}[\![\ t_1\ ]\!] \ldots \mathcal{N}[\![\ t_n\ ]\!]\} \\
\mathcal{N}[\![\ g\, t_0 \ldots t_n\ ]\!] & = & 1 + \max\{\mathcal{N}[\![\ t_0\ ]\!] \ldots \mathcal{N}[\![\ t_n\ ]\!]\}
\end{array}$$

□

LEMMA **10** *Suppose that $\mathcal{F}$ is applied to a function term and a restricted treeless program. Then for each transformation step $\mathcal{F}[\![\ t\ ]\!] = \ldots \mathcal{F}[\![\ t_i\ ]\!] \ldots$, $\forall i, \mathcal{N}[\![\ t_i\ ]\!] \leq \mathcal{N}[\![\ t\ ]\!]$.* □

PROOF: Induction on i proceeding by cases on t. □

DEFINITION **29** Define terms in *fct* by:

$$\begin{array}{lcl}
fct & ::= & ft \mid ef[c\, ft_1 \ldots ft_n] \\
ef & ::= & [] \mid g\ ef\, ft_1 \ldots ft_n
\end{array}$$

where ef is a function context. □

LEMMA **11** *Suppose that $\mathcal{F}$ is applied to a function term and a restricted treeless program. Then for each transformation step $\mathcal{F}[\![\ t\ ]\!] = \ldots \mathcal{F}[\![\ t_i\ ]\!] \ldots$, $\forall i, t \in \text{fct} \Rightarrow t_i \in \text{fct}$.* □

PROOF: Induction on i proceeding by cases on t. □

PROOF:(of the Deforestation Theorem) Consider the initial call $\mathcal{F}[\![\ t\ ]\!]$ for some $t \in ft \subseteq fct$ with $\mathcal{N}[\![\ t\ ]\!] = k$. Any term t' encountered in a recursive call is contained in the syntactic class *fct* by Lemma 11, and therefore contain 0 or 1 constructor from p. Further, by Lemma 10, $\mathcal{N}[\![\ t'\ ]\!] \leq k$. Since there are only finitely many terms t' containing functions, variables, 0 or 1 constructor from p with $\mathcal{N}[\![\ t'\ ]\!] \leq k$, only finitely many terms can be encountered in recursive calls. Since $\mathcal{F}$, and thereby $\mathcal{D}$, encounters only finitely many different terms, $\mathcal{F}$ terminates by the Quasi-termination Theorem for

$\mathcal{D}$ (proved for $\mathcal{G}$ in Chapter 6.[2]) □

Besides ensuring termination, the two lemmas provide insight to the interior of $\mathcal{F}$. If we run $\mathcal{F}$ on an ft-term, execution will proceed as follows. First the term is decomposed into a context and redex of some type; then the redex is unfolded repeatedly (possibly through necessary instantiations) until finally we arrive at a term containing a constructor, which will then be outermost; now the surrounding g-call will be the redex, and is unfolded repeatedly until a constructor is finally produced. This is continued until the constructor is propagated all the way to the root; then the algorithm starts on the constructor arguments; at this point the overall term will still have a nesting at most equal to the nesting of the original term, and none of the constructor arguments can contain a constructor, so the constructor arguments will be ft-terms. This closely resembles the desired behaviour we described in Section 3.2.

In conclusion, $\mathcal{F}$ works well when supplied restricted treeless functions: it returns a treeless term (which contains no intermediate structures), and it terminates.

It seems to be foklore in the community that the Deforestation Theorem can be extended to the following slightly more general form (although we have not seen a proof of it):

THEOREM **11** *(Folklore Deforestation Theorem) For an arbitrary term t and a treeless program p, $\mathcal{F}[\![\, t \,]\!]$ terminates.* □

Section 10.6 shows that our method (using one of the extensions described in the preceding chapter) includes Wadler's in the sense that applying our grammar construction to an arbitrary term and a treeless program will yield a resulting grammar with no infinity, thereby showing that no annotations need be put on the program. This incidentally yields a proof (although a rather complicated one) of the above theorem. Also, there are examples of non-treeless programs which do not need annotations by our method (see the end of this chapter.)

[2]This last point was not mentioned in [Fer88]; it is non-trivial since $\mathcal{F}$ does not recall *all* terms, *c.f.* Chapter 6.

10.2 Wadler's blazed treeless terms

We have already in Chapter 4 developed a means of annotating terms to inform the deforestation algorithm to leave certain terms untouched. Readers familiar with partial evaluation will have recognized our annotations as something similar to binding time annotations of partial evaluation. However, in the context of deforestation, the means of annotation was introduced in [Wad88] under the name of blazing. This section describes the invention. Apart from being historically interesting, the present section lays the ground for the next section.

To explain the origin of blazing, we assume that our language has a Milner type discipline [Mil78] with integers and booleans as base values (primitive 0-ary constructors), and a number of base functions on these types that we have no means of evaluating at $\mathcal{F}$-time.

Now consider the program:

$$\begin{array}{lcl} sum\ xs & = & sum'\ xs\ 0 \\ sum'\ Nil\ a & = & a \\ sum'\ (Cons\ x\ xs)\ a & = & sum'\ xs\ (a+x) \end{array}$$

Here, sum' and sum aren't restricted treeless, because of the $+$ in the last clause and 0 in the first, respectively. Here is what happens when we run $\mathcal{F}$ on $sum\ xs$: [3]

$$\begin{array}{lcl} \mathcal{F}[\![\ sum\ xs\ ]\!] & = & \mathcal{F}[\![\ sum'\ xs\ 0\ ]\!] = g_1\ xs \\ \quad \text{where} & & \\ \quad g_1\ (Cons\ x'\ xs') & = & \mathcal{F}[\![\ sum'\ (Cons\ x'\ xs')\ 0\ ]\!] \\ & = & \mathcal{F}[\![\ sum'\ xs'\ (0+x')\ ]\!] \\ & = & g_2\ xs'\ x \\ \qquad \text{where} & & \\ \qquad g_2\ x'\ (Cons\ x''\ xs'') & = & \mathcal{F}[\![\ sum'(Cons\ x''\ xs'')(0+x')\ ]\!] \\ & = & \mathcal{F}[\![\ sum'\ xs''((0+x')+x'')\ ]\!] \end{array}$$

and so on.

So $\mathcal{F}$ gets into problems on this program and term. By allowing terms where we previously required variables, the term we are rewriting no longer

[3] In a simplified manner; new functions are only introduced when necessary for pattern matching, and some clauses of new definitions are omitted.

conforms to *fct*; and the nesting may grow arbitrarily, resulting in infinite transformation. If 0 and $(a + x)$ were somehow regarded as atomic entities, *i.e.* variables, the rewriting of g_1 would have discovered, that we had already seen $sum'\, xs'\, (0 + x')$ as $sum'\, xs\, 0$ instead of giving rise to g_2..

Since integer-valued terms are not constructed and destructed the same way as trees with proper constructors, no gain in effeciency can be expected from transforming such terms. So $\mathcal{F}$ might as well treat the integer-valued terms above as variables. Thus, doing this is both *necessary* to ensure termination, and *desirable* since there is no possible gain in efficiency.

So [Wad88] needed to invent two things: a way of annotating a term to inform $\mathcal{F}$ which subterms should be regarded atomic, and a precise way of making $\mathcal{F}$ follow these annotations. As we saw, the above transformation would terminate with such a scheme.

Thus inspired, [Wad88] lets any term t of type *int* or *bool* be annotated with a $\ominus$ and goes on with the following definition

DEFINITION **30** (Blazed treelessness) Define a program, p, to be *blazed treeless* if every right hand side of every definition of p conforms to *btt*:[4]

$$\begin{array}{lcl} btt & ::= & vv \mid c\, btt_1 \ldots btt_n \mid f\, vv_1 \ldots vv_n \mid g\, vv_0 \ldots vv_n \\ vv & ::= & v \mid (c\, vv_1 \ldots vv_n)^{\ominus} \mid (f\, vv_1 \ldots vv_n)^{\ominus} \mid (g\, vv_0 \ldots vv_n)^{\ominus} \end{array}$$

Each right hand side of p is called a blazed treeless term if p is blazed treeless. □

A blazed treeless term is the same as a treeless term, except that where we previously required variables, we now allow a *vv* term. A *vv* term is a variable, or a blazed term with a $\ominus$ on it and a $\ominus$ on all non-variable subterms.

Recall that $\mathcal{F}$ terminated with treelees terms as input. Therefore, it seems clear that with a blazed treeless term as input, a version of the deforestation algorithm treating annotated subterms as variables will terminate as well. This latter version of the deforestation algorithm was invented in [Wad88] and is essentially our $\mathcal{G}$. Specifically, a function call in the redex is extracted

[4] Unannotated terms in the definition correspond to terms in [Wad88] annotated with $\oplus$.

iff it has a $\ominus$ on it, and arguments in a call in the redex are extracted iff they have a $\ominus$ on them.[5]

With these mechanisms, [Wad88] asserted:

THEOREM **12** *(Blazed Deforestation Theorem.) For an arbitrary term t and a blazed treeless program p, $\mathcal{G}[\![\ t\ ]\!]$ terminates.* □

REMARK **8** Further assertions similar to those in Remark 7 were also made, see [Wad88] for the details. □

As an example, consider the above example term. First its subterms of type *int* are annotated. Fortunately, this yields a blazed treeless term, so we can run the generalizing deforestation algorithm on it. This yields:

$$
\begin{array}{lcl}
\mathcal{F}[\![\ sum\ xs\ ]\!] & = & \mathcal{F}[\![\ sum'\ xs\ 0^{\ominus}\ ]\!] \\
 & = & let\ v = \mathcal{F}[\![\ 0\ ]\!]\ in\ \mathcal{F}[\![\ sum'\ xs\ v\ ]\!] \\
\quad \text{where} & & \\
\mathcal{F}[\![\ 0\ ]\!] & = & 0 \\
\mathcal{F}[\![\ sum'\ xs\ v\ ]\!] & = & g\ xs\ v \\
\quad \text{where} & & \\
\quad g\ Nil\ v & = & \mathcal{F}[\![\ v\ ]\!] = v \\
\quad g\ (Cons\ x'\ xs')\ v & = & \mathcal{F}[\![\ sum'\ xs'\ (v + x')^{\ominus}\ ]\!] \\
 & = & let\ v' = \mathcal{F}[\![\ v + x'\ ]\!]\ in\ \mathcal{F}[\![\ sum'\ xs'\ v'\ ]\!] \\
\qquad \text{where} & & \\
\qquad \mathcal{F}[\![\ v + x'\ ]\!] & = & v + x' \\
\qquad \mathcal{F}[\![\ sum'\ xs'\ v'\ ]\!] & = & g\ xs'\ v'
\end{array}
$$

In the last step $\mathcal{F}$ discovered that it was not necessary to transform the overall term, because it had already been seen.

10.3 Chin's extended treeless terms

The results of the preceding section could still not be used to apply $\mathcal{G}$ to arbitrary programs and terms, because the annotation was carried out according

[5] Actually, the extraction mechanism in [Wad88] extracts not only calls from the redex and arguments from the call in the redex, but arbitrary annotated subterms; the difference seems insignificant.

to the type of the subterms. We simply had to hope that the annotated term was a blazed treeless term; otherwise termination could not be guaranteed.

However, one might dispense with the type based annotation scheme and instead reason as follows. In Section 10.1 we saw that $\mathcal{F}$ terminates for treeless programs. In Section 10.2 we saw that by annotating certain subterms not having the treeless form with a minus, termination could be guaranteed. But why not use this as a general principle: given an arbitrary program, annotate all subterms not having the treeless form with a $\ominus$. This was done by Chin in [Chi90] and is described in this section.

DEFINITION **31** (Extended treelessness.) Define a program p to be *extended treeless* if every right hand side of every definition in p conforms to ett:[6]

$$\begin{array}{lcl} ett & ::= & v \mid c\ ett_1 \ldots ett_n \mid f\ a_1 \ldots a_n \mid g\ a_0 \ldots a_n \\ a & ::= & v \mid ett^{\ominus} \end{array}$$

Each right hand side of p is called extended treeless if p is extended treeless. □

Extended treeless terms are somewhat similar to blazed treeless terms, but the annotations are no longer made according to types, and subterms of annotated terms need no longer be annotated. By extracting annotated calls from the redex and annotated arguments from the call in the redex, one can apply $\mathcal{G}$ to an extended treeless term; [Chi90] proved:

THEOREM **13** *(Extended Deforestation Theorem) For an arbitrary term t and an extended treeless program p, $\mathcal{G}[\![\ t\ ]\!]$ terminates.* □

REMARK **9** The further assertions similar to those for treeless terms in Remark 7 and blazed treeless terms in Remark 8 are as follows. Under the conditions of the preceding theorem, (1) the result of the transformation is an extended treeless term and new extended treeless functions; (2) assuming, in addition, that whenever there is a call to a function which is non-linear in its i'th parameter, the actual parameter is annotated with $\ominus$, it is the case that the resulting term is at least as efficient as the original. □

[6] Unannotated terms in the definition correspond to terms in [Chi90] annotated with $\oplus$.

Given an arbitrary program p it is easy to turn p into an extended treeless program: annotate every non-variable argument with $\ominus$. Further, the condition in (2) above can be satisfied simply by putting the necessary annotations on calls in the term and program. We call this *Chin's Polyvariant annotation scheme.*

So Chin's Polyvariant annotation scheme allows us to apply $\mathcal{G}$ to an arbitrary term and program with guaranteed termination and non-degradation of efficiency.

10.4 Further ideas by Chin

The description in the preceding section represents a simplified picture of Chin's method as described in his works. Specifically, the technique can be improved by using a bottom-up transformation technique which we describe now.

By the scheme of the preceding section, the program will get many annotations while the term will receive only few. One can take advantage of this fact as follows.

First construct a *call tree.* Each node contains the name of some functions. The names of mutually recursive functions are put in the same node. If some function with name f calls a function with name g (where f and g are not in the same node) the tree contains a direced edge from the node containing f to the node containing g. Now we carry out a process on the nodes bottom-up, *i.e.* always on the node containing g before on the node containing f in the example above.

For a node containing the function names $h_1 \ldots h_n$, the process runs as follows. First make a *copy* of each of the right hand sides of the definitions of $h_1 \ldots h_n$. Now we consider each of these as the term we wish to transform, and the program is the original definitions of $h_1 \ldots h_n$. Now annotate the program and term as described above, and run $\mathcal{G}$ on each of the terms.

Suppose that the overall term we wish to transform is t and p is the program. The idea above has two advantages. First, if t has a call to f, then the call will be replaced by f's body which will then be transformed. If t has several calls to f, possibly via other functions, this will happen several times. By transforming bottom-up, the body of f will only be optimized once, although of course the optimized body of f still will be encountered

several times from t. The second advantage is that more transformation may be possible because transformation of a set of mutually recursive functions may reduce the need for annotations higher up in the call tree, *e.g.* if a non-linear function is transformed into a linear function.

We should also mention that not all violations against the treeless form are dangerous. For instance, if the program consists of two functions f and g where f calls g and g calls itself, then it is not a problem if f calls g with a non-variable argument. Chin's *Double annotation scheme* takes advantage of this fact, see [Chi93a].

Finally, Chin has many improvements of the basic technique above, which are, however, beyond the scope of this report; we refer the reader to [Chi93a].

10.5 Previous semantic analyses

We are only aware of one previous semantic analysis with the purpose of ensuring termination of deforestation, namely that described in [Ham91], which used analyses described in [Ham90], [Ham92a], [Ham92b]. However, the original method turned out *not* to ensure termination [Ham92c]. Using some of the same ideas as in the original article, the method was subsequently modified to ensure termination [Ham92c]. We have not yet investigated the connection between this new method and our method.

10.6 On the relation between our method and previous methods

In preceding chapters we have developed a rather complicated method of ensuring termination of $\mathcal{G}$. Compared to our method, Wadler's syntactic criteria may be checked efficiently, and Chin's annotations can be calculated efficiently. We need, therefore, to show that our method yields an improvement over Wadler' and Chin's (Polyvariant) method.

In our perception the essence of Chin's method as descibed in his works is what we have described in Section 10.2. The essence of what we described in that section is that treeless programs are harmless, and the method essentially turns non-treeless programs into (extended) treeless programs.

We show in this section that given an arbitrary term t and a treeless program p, for the grammar $< C, D >$ computed from t and p there is no $N^f \to e[N^f] \in C^c$ with $e \neq []$ and no $N^v \to e(N^v) \in D^d$ with $e \neq ()$. This implies that no annotations will be put on the program.

This means that any program and term that Wadler's method can handle, will not receive any annotations by our analysis. This does not yield a rigorous proof that our method strictly extends Chin's methods, but we do feel that our method through suitable extensions, specifically those of the previous chapter, can be suited to find annotations no worse than those found by Chin's methods.

Before proceeding to the theorem alluded to above, we recall the technique of tracing top-level arguments. Each function call was given a unique number and we took n copies of the program, where in the i'th copy all function names in definitions and calls were subscripted with i. In the following theorem we assume that the numbering of the term has been carried out in such a manner that whenever a function call occurs within the argument of another function call in the initial term, the function name of the former call has a lower number. Of course this is always possible: viewing the term as a tree we assign 1 to the root, then $2\ldots$ to its immediate children, then numbers to their children, *etc.*

When a function name or nonterminal has been assigned a number n we say that the function or nonterminal is at level n. Deeper level means higher n.

THEOREM **14** *Let p be a treeless program and t an arbitrary term. Let $< C, D >$ be the grammar computed from p and t using the extension of tracing top-level arguments. Then there is no $N^f \to e[N^f] \in C^c$ with $e \neq []$ and no $N^v \to e(N^v) \in D^d$ with $e \neq ()$.* $\Box$

PROOF: We prove by induction on the number of steps of the grammar algorithm that all the productions in $U_{\mathcal{D}}^k$ and $V_{\mathcal{D}}^k$ have the following forms, which we call *treeless productions:*

1. $N^h \to c\ tt_1 \ldots tt_n$ where tt_i is a treeless grammar term possibly containing nonterminals of form $N^{v_i^h}$ on the same level as N^h.

2. $N^h \to N^{v_i^h}$, where N^h and $N^{v_i^h}$ are on the same level.

3. $N^h \to N^{h'}$, where N^h and $N^{h'}$ are on the same level.

4. $N^h \to g\ N^{v^h_{i0}}\ N^{v^h_{i1}} \ldots N^{v^h_{in}}$, where N^h, g and all the $N^{v^h_{il}}$ are on the same level.

5. $N^h \to \bullet$.

6. $N^{v^h_i} \to \bullet$.

7. $N^{v^h_i} \to N^{v^{h'}_j}$, where $N^{v^h_i}$ and $N^{v^{h'}_j}$ are on the same level.

8. $N^{v^h_i} \to t$ where all the function names and nonterminals in t are on strictly deeper levels than the level of $N^{v^h_i}$, and where all arguments of function calls in t are on a strictly deeper level than the function name in the call.

9. $N^0 \to t$ where all arguments of function calls in t are on a strictly deeper level than the function name in the call.

$k = 0$: true by our numbering scheme.

$k = j + 1$: Applying $\diamond$ to productions of the above form yields the above set of productions and three new kinds:

1. $N^h \to g\ N^v\ N^{v^h_{i1}} \ldots N^{v^h_{in}}$, where N^h, g and all the $N^{v^h_{il}}$ are on the same level.

2. $N^h \to g\ \bullet\ N^{v^h_{i1}} \ldots N^{v^h_{in}}$, where N^h, g and all the $N^{v^h_{il}}$ are on the same level.

3. $N^h \to g\, c\, t_1 \ldots t_k\, N^{v^h_{i1}} \ldots N^{v^h_{in}}$, where N^h, g and all the $N^{v^h_{il}}$ are on the same level, and where all the function names and nonterminals in t are on strictly deeper levels than the level of $N^{v^h_i}$, and where all arguments of function calls in t are on a strictly deeper level than the function name in the call.

Now verify for each of these kind of productions (including the original treeless productions) that the functions $\mathcal{U}_\mathcal{D}$ and $\mathcal{V}_\mathcal{D}$ yield only treeless productions.

From this we easily find that C and D contain only treeless productions.

Finally note that if all the productions of C and D are treeless, there cannot be a production $N^h \to e[N^h]$ with the two nonterminals on the same level and $e \neq []$ in U and there cannot be a production $N^v \to e[N^v]$ with the two nonterminals on the same level and $e \neq ()$ in V. $\Box$

It only remains to show that there are non-treeless programs which do not receive annotations by our method. The following example shows the core of this phenonenom.

Consider the term and program:

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

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

The control grammar is:

$$\begin{array}{lcl} N^0 & \to & f\ x \mid N^f \\ N^f & \to & f'\ (C\ N^x) \mid N^{f'} \\ N^{f'} & \to & g\ N^y \mid N^g \\ N^g & \to & f\ N^z \mid N^f \\ N^y & \to & c\ N^x \end{array}$$

The data grammar is:

$$\begin{array}{lcl} N^x & \to & \bullet \mid N^z \\ N^z & \to & N^x \\ N^y & \to & c\ N^x \end{array}$$

Note that these grammars do not contain infinity.

In the control grammar we can see that the analysis reasons as follows. First, f calls f' (signified by the production $N^f \to N^{f'}$.) In the program we can see that the argument is larger than that which f itself received. Second, f' calls g ($N^{f'} \to N^g$.) The argument is the same as that which f' received. Finally, g calls f ($N^g \to N^f$.) Here the argument is smaller than that which g itself received. The point is: will the argument to f get bigger and bigger in each incarnation of f, or is the decrement by g significant

enough to outweigh the increment by f? Well, in the data grammar we see that our analysis discovered that the argument that g binds to z when it is called from f' is the argument which f originally received ($N^z \to N^x$) and g then calls f with that same argument ($N^x \to N^z$), and this does not yield larger and larger arguments.

Note that the program is non-treeless. The reader familiar with Chin's work may also note that his methods would classify f' as an unsafe consumer, and extract the argument $C\ x$. This would in fact prevent the C in the call to f' from being eliminated by g. It is not hard to imagine realistic examples where this is significant. It is not hard to generalize this phenonenom to a class of programs.

Chapter 11

Conclusion

The first section describes related work that was not mentioned in Chapter 10. This section has the traditional handwaving character. The second section outlines directions for further research. The last section concludes.

11.1 Other related work

We have already a few times come across the work of Holst [Hol91] which is concerned with termination of partial evaluation. In this work the binding time annotations are made on the basis of observations of so-called *increasing* and *decreasing parameters.*

It seems that there are many similarities in the ideas. However, it presently seems to the author that the notion of *e.g. increasing* parameter is more similar to Chin's notion of an accumulating parameter (a non-variable argument) than to our notion of an accumulating parameter (a parameter v for which $N^v \rightarrow e(N^v) \in D^d$.) It would be interesting to analyze the connections more precisely.

This would be particularly interesting if our method were to be extended to approximate a transformation which combines deforestation with partial evaluation. In this connection it also seems interesting to investigate whether base functions such as + and primitive 0-ary constructors for numbers can be incorporated into the deforestation algorithm in a satisfactory manner via the view mechaninsm [Wad87a]. For reasons of efficiency, programming + as a user function and incorporating natural numbers as constructors directly

does not seem satisfactory.

A transformation algorithm doing both deforestation and partial evaluation is likely to be similar to Turchin's supercompiler which performs these transformations. Therefore, a method of ensuring termination for the former transformation should be compared to Turchin's on-line strategy [Tur88] for his supercompiler.

11.2 Directions for further research

First of all the method should be implemented and experimental results for it should be compared with other experimental results. We are only presently aware of the experimental results in [Gil93].

The grammar algorithm as we have stated in in this paper seems rather inefficient. However, the basic fix point technique may be improved by observing that certain productions are "dead" *i.e.* once they have occurred as argument of the functions which extends the grammar, they will never again add any new productions. Other similar observations may improve the efficiency of the grammar algorithm significantly.

In the preceding section we considered the extension of our method to a transformation that does in addition partial evaluation. The method could also be changed to other fold/unfold transformation techniques, for instance *Tupling* [Chi93a], [Chi93b]. Also, the method could be extended to a more powerful language including general patterns or higher-order functions. In the latter case, the work should be compared to [Chi92a], [Mar92] and [Ham93].

11.3 Conclusion

We have described a means of ensuring termination of the deforestation algorithm which strictly extends (some) previous methods. We believe that extension of previous methods can be beaten by extensions of our method. Compared to previous methods, our means of ensuring termination is rather inefficient.

The applicability of our method compared to previous methods must ultimately depend on its behaviour on "real" programs.

Bibliography

[Aho86] Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools* . Addison-Wesley, 1986.

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

[Aug85] Lennart Augustsson. Compiling Lazy pattern-matching. In *Conference on Functional Programming and Computer Architecture.* Lecture Notes in Computer Science 201, 1985.

[Bir80] R.S. Bird. Tabulation Techniques for Recursive Programs. In *Computing Surveys.* Vol. 12, No 4, December 1980.

[Bir88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[Blo88] Adrienne Bloss, P. Hudak, J. Young. An Optimising Compiler for a Modern Functional Language. In *The Computer Journal.* vol 31. no 6, 1988.

[Bon90] Anders Bondorf. *Self-Applicable Partial evaluation.* Ph.D. thesis, DIKU-Rapport 90/17, Departement of Computer Science, University of Copenhagen, 1990.

[Bur77] R. M. Burstall & John Darlington. A Transformation system for Developing Recursive Programs. In *Journal of the ACM.* Vol. 24, No. 1. January 1977.

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

[Chi92a] Wei-Ngan Chin. Fully Lazy Higher-Order Removal. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Yale University, 1992.

[Chi92b] Wei-Ngan Chin. Safe Fusion of Functional expressions. In *ACM Lisp and Functional Programming Conference.* San Francisco, California, June 1992.

[Chi93a] Wei-Ngan Chin. Safe Fusion of Functional expressions. In *Journal of Functional programming.* Special issue, 1993. *TO APPEAR.*

[Chi93b] Wei-Ngan Chin. Towards an Automated Tupling Strategy. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Copenhagen, Denmark, 1993..

[Chi93c] Wei-Ngan Chin. *A Modular strategy for Combining the Fusion and Tupling Methods.* Unpublished manuscript. 1993.

[Coh83] Norman H. Cohen. Eliminating Redundant Recursive Calls. In *ACM Transactions on Programming Languages and Systems.* Vol. 5 No 2, April 1983.

[Cou86] Bruno Courcelle. Equivalences and Transformations of Regular Schemes — Applications to Recursice Program Schemes and Grammars. In *Theoretical Computer Science.* 42, 1986.

[Dar81] John Darlington. An Experimental Program Transformation and Synthesis System. In *Artificial Intelligence.* 16, 1981.

[Der87] Nachum Dershowitz. Termination of Rewriting. In *Journal of Symbolic Computation.* 3, 1987.

[Der90] Nachum Dershowitz & Jean-Pierre Juunnaud. Rewrite Systems. In *Handbook of theoretical Computer Science.* 1990.

[Fea82] Martin S. Feather. A System for Assisting Program Transformation. In *ACM Transaction on Programming Languages and Systems.* 4(1), 1982.

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

[Gil93] Andrew Gill, John Launchbury & Simon L Peyton Jones. A Short Cut to Deforestation. In *Conference on Functional Programming and Computer Architecture*. Copenhagen, Denmark, 1993.

[Glu89] Robert Glück & V. F. Turchin. *Experiments with a Self-Apllicable Supercompiler.* Technical Report, City University, New York, 1991.

[Glu91] Robert Glück. Towards Multiple Self-Application. In *ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation.* 1991.

[Glu92b] Robert Glück. *On the Generation of Specializers.* Unpublished manuscript. 1992.

[Glu92] Robert Glück & Andrei Klimov. Ockham's razor in metacomputation:the notion of a perfect process tree. *3rd International Workshop on Static analysis, Padova, Italy.* 1993, TO APPEAR

[Glu93b] Robert Glück & Jesper Jørgensen. *Generating Optimizing Specializers.* Unpublished manuscript. 1993.

[Gom90] Carsten K. Gomard & Neil D. Jones. *Compiler Generation By Partial Evaluation: A case Study.* DIKU-rapport 90/16, Department of Computer Science, University of Copenhagen, 1990.

[Ham90] G. W. Hamilton & S. B. Jones. *Compile-Time Garbage Collection by Necessety Analysis.* Technical Report Department of Computing Science and Mathematics University of Stirling, Scotland, 1990.

[Ham91] G. W. Hamilton & S. B. Jones. Extending Deforestation for First Order Functional Programs. In *1991 Glasgow Workshop on Functional Programming.* 1991.

[Ham92a] G. W. Hamilton. *Sharing Analysis of Lazy First Order Functional Programs,.* Unpublished manuscript. 1992.

[Ham92b] G. W. Hamilton. *Compile-Time Optimisation of Storage Utilisation for Lazy First Order Functional Programs.* Unpublished manuscript. 1992.

[Ham92c] G.W Hamilton. *Correction to [Ham91].* e-mail correspondence. 1993.

[Ham93] G. W. Hamilton. *Higher Order Deforestation.* Unpublished manuscript. 1991.

[Har78] Michael A Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[Hol91] Carsten Kehler Holst. Finiteness Analysis. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, LNCS 523.* Cambridge, Massachussets, 1991.

[Hug88] John Hughes. Backwards Analysis of Functional Programs. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjrner & N. D. Jones, North-Holland, 1988.

[Hug90a] John Hughes. Why Functional Programming Matters. In *Research topics in Functional Programming.* Ed. D. Turner, Addison-Wesley, 1990.

[Hug90b] John Hughes. Compile-Time Analysis of Functional Programs. In *Research topics in Functional Programming.* Ed. D. Turner Addison-Wesley, 1990.

[Jon81] Neil D. Jones & Steven S. Mechnick. Flow Analysis and Optimization of LISP-like Structures. In *Steven S. Muchnick & Neil D. Jones (Ed.) Program Plow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[Jon86] Neil D. Jones & Alan Mycroft. Data flow Analysis of Applicative Programs Using Minimal Function Graphs: Abridged Version. In *13th ACM Symposium on Principles of Programming Languages.* St. Petersburg, Florida, 1986.

[Jon87] Neil D. Jones. Flow analysis of Lazy higher-order functional programs. In *Abstract Interpretation of Declarative Languages.* Eds. Samson Abramsky & Chris Hankin, Ellis Horwood, London, 1987.

[Jon88] Neil D. Jones. Automatic Program Specialization: A re-examination from basic principles. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjørner & N. D. Jones, North-Holland, 1988.

[Jon91] Neil D. Jones, Peter Sestoft, Harald Søndergaard. *Mix. A Self-applicable Partiel Evaluator for Experiments in Compiler Generation.* DIKU-rapport 91/12, Department of Computer Science, University of Copenhagen, 1991.

[Jon92] Neil D. Jones, Carsten K. Gomard & Peter Sestoft. *Partial evaluation and Automatic Program Generation.* Prentice-Hall, 1993..

[Kot??] Unfold/Fold Transformations,. *??.* ??.

[Lau91] John Launchbury. A Strongly-Typed Self-Applicable Partial Evaluator. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, LNCS 523.* Cambridge, Massachussets, 1991.

[Mal93] Karoline Malmk: Towards Efficient Partial Evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Copenhagen, Denmark, 1993.

[Mar92] S. Marlow & P.L. Wadler. Deforestation for higher-order functions. In *Functional Programming, Glasgow 1992.* Ed. J. Launchbury, Workshops in Computing, 1992.

[Mei91] Erik Meijer, Maarten Fokkinga & Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, LNCS 523.* Cambridge, Massachussets, 1991.

[Mil78] R. Milner. A therory of type polymorphism in programming. In *Journal of Computer and System Sciences.* 17, 1978.

[Mog88] Torben Mogensen. Partially Static Structures in a Self-applicable Partial Evaluator. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjrner & N. D. Jones, North-Holland, 1988.

[Mog91] Torben Mogensen. *Variants of Unfold/fold Strategies.* Unpublished manuscript. (In Danish.) 1991.

[Nel91] G.C. Nelan. *Firstification.* Ph.D Dissertation, Arizona State University.

[Run89] Colin Runciman, Mike Firth, Nigel Jagger. Transformation in a Non-Strict Language: An approach to instantiation. In *1989 Glasgow Functional Programming Workshop.* 1989.

[Ses88] Peter Sestoft. Automatic Call Unfolding in a Partial Evaluator. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjrner & N. D. Jones, North-Holland, 1988.

[She93] Tim Sheard & Leonidas Fegaras. A Fold for All Seasons. In *Conference on Functional Programming and Computer Architecture.* Copenhagen, Denmark, 1993.

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

[Tur80] Valentin F. Turchin. Semantic Definitions in Refal and Automatic Production of Compilers. In *LNCS 94.* 1980.

[Tur80b] Valentin F. Turchin. *The Language REFAL—The Theory of Compilation and Metasystem Analysis.* Courant Computer Science Report 20, 1980.

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

[Tur86] Valentin F. Turchin. The Concept of a Supercompiler. In *ACM Transactions on Programming Languages and Systems.* Vol. 8, No. 3, 1986.

[Tur86b] Valentin F. Turchin. *Refal: A Language for Linguistic Cybernetics.* Technical Report, City University, New York, 1986.

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

[Tur92] Valentin F. Turchin. Program Transformation by Metasystem Transitions. Submitted to *Journal of Functional Programming*. 1993.

[Tro88] A.S. Troelstra & D. van Dalen. *Constructivism in Mathematics, an introduction, vol.1*. Studies In Logic And The Foundations Of Mathematics, vol 121. North-Holland, Amsterdam, 1988.

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

[Wad85] P. L. Wadler. Listlessness is better than lazyness II: Composing Listless functions. In *Workshop on Programs as Data objects*. Lecture notes in Computer Science 217, Copenhagen, 1985.

[Wad87a] P. L. Wadler. Views: A way for pattern-matching to cohabit with data abstraction. In *14th Conference on Principles of Programming Languages*. 1987.

[Wad87b] P. L. Wadler. Efficient compilation of pattern-matching. In *The implementation of Functional Programming Languages,*. Ed. S. L. Peyton Jones, Prentice-Hall, 1987.

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

[Wat91] Richard Waters. Automatic Transformation of Series Expressions into Loops. In *ACM Transactions on Programming Languages and Programming Systems (TOPLAS)*. 1991.

[Wei91] Daniel Weise, Roland Conybeare, Erik Ruf, Scott Seligman. Automatic Online Partial Evaluation. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*. Cambridge, Massachussets, 1991.