Driving-based Program Transformation in Theory and Practice

Ph.D. Thesis

Jens Peter Secher

Department of Computer Science
Faculty of Science
University of Copenhagen

September 16, 2002

Contents

Abstract

This thesis deals with driving-based program-transformation techniques for first-order, non-strict functional programming languages.

The first part of the thesis describes a selection of existing term-based program transformers in a uniform framework, touching upon well-known techniques for speculative execution, program specialisation, term rewriting, and measures for ensuring termination.

The second part of the thesis presents a fully automatic program transformer which improves existing techniques by rephrasing driving-based program transformation in a setting where terms are represented as acycylic directed graphs (dags). The dag-based setting makes it possible for the transformer to eliminate duplication of computation, intermediate data structures, and multiple traversals of data structures, which results in transformed programs that might be asymptotically faster than their original counterparts.

The dag-based setting also leads to the development of a *parameterised* operational semantics of orthogonal term-rewriting systems, in the sense that the underlying graph machinery can freely be chosen from within a wide range of possible implementation strategies without affecting the observable behaviour of programs. Three implementations are shown, namely *call-by-name*, *call-by-need*, and *collapsed-jungle evaluation*.

A notion of dag-based grammars with an expressiveness that lies on border between context-free and context-sensitive string grammars is developed, and these grammars are used to give a concise representation of the input–output behaviour of programs, illustrated by an automatic program inverter.

Lastly, an interesting connection between dag-based program transformation of simple constraint checkers and binary-decision diagrams is established and used to solve the problem of interactive product-line configuration.

Keywords and phrases: Program transformation, non-strict functional languages, orthogonal term-rewrite systems, directed acyclic graphs, deforestation, tupling, supercompilation, well-quasi-orders, program inversion, dag grammars, binary-decision diagrams, product-line configuration.

Preface

In this thesis, submitted in partial fulfilment of the requirements for the Danish Ph.D. degree, I have tried to give a coherent description of the research I have carried out between May 1999 and June 2002 under the supervision of Morten Heine Sørensen and Neil D. Jones at the Departmen of Computer Science, University of Copenhagen (DIKU).

Although certain areas of my work (e.g., [3, 25]) are not included in the following chapters, this thesis comprises the bulk of research I have carried out. Chapter 6 is based on a paper [97] I presented in July 1999 at the *Third International Conference on Perspectives of System Informatics.* Chapters 8 and 9 are based on a paper [95] I presented in May 2001 at the *Second Symposium on Programs as Data Objects.* Chapter 11 is an extended version of a paper [96] I presented in January 2002 at the *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Chapter 12 is based on a paper [107] to be included in a Festschrift in honour of Neil D. Jones.

The background chapters in part one were developed for a course held at DIKU in the Fall of 1999. Parts of these chapters have been used by Morten Heine Sørensen and Michael Leuschel at the European Summer School in Logic, Language and Information, Utrecht in the summer of 1999. Chapters 2, 4 and 7 owe much to a set of notes developed by Morten Heine Sørensen for a graduate course he held together with Michael Leuschel during Fall 1997. Chapter 5 owes much to a paper made by Morten Heine Sørensen and Robert Glück [106].

The above account of the origins of the chapters reveal that I am indebted to Morten Heine Sørensen, Neil D. Jones, Robert Glück and Michael Leuschel for guidance and inspiration. Moreover, people with whom I have had discussions somehow related to program tranformation, the main theme of this thesis, include Stephen Alstrup, Peter Holst Andersen, Niels Christensen, Michael Florentin, Arne Glenstrup, Jörgen Gustavsson, Inge Li Gørtz, Carsten Kehler Holst, Jesper Jørgensen, Armin Kühnemann, David Lacey, Ken Friis Larsen, Torben Mogensen, Jesper Møller, Peter Møller Neergaard, Oege de Moor, Christian Worm Mortensen, Jean-Yves Moyen, Chin Soon Lee, Sergei Romanenko, Peter Sestoft, Ganesh Sittampalam, German Vidal.

Special thanks go to Henning Niss and Henning Makholm for valuable discussions about everything from sour dough over LaTeX to the λ-cube.

Thanks to my whole family for supporting me, accepting “No, I have to work” rather often. Especially my wife and children beared with my bad moods and have heard “No, daddy has to work” too many times.

I have used a lot of special notation in this thesis. Appendix A describes this notation, and there is a symbol index at the very last pages of this thesis.

JPS, May 2002

Production notes: This thesis has been typeset on a Debian GNU/Linux system using the teTEX distribution of LATEX. For the main text, I use the Concrete typeface designed by Donald E. Knuth, and for math I use the AMS Euler typeface designed by Hermann Zapf, together with a set of script letters designed by Ralph Smith.

Chapter 1

Introduction

This thesis is concerned with *transformation of functional programs*. The distinctive features of such programs is that they lack *side-effects*. Put crudely: they have no *assignment* command. Examples of such languages include Haskell, a non-strict functional language; Mercury, a non-strict functional-logic language; and ML and Lisp, which are strict functional languages. The latter two languages are not entirely free of side-effects, but the core of each language is.

Folk law tells us that functional programming languages are *slow* and that C programs usually run much faster than e.g., Haskell programs, but in recent years, researchers have addressed such compiler problems and several compilers for functional languages have reached a state where they can keep up with good C compilers; see for instance how OCaml compares or even outperforms C in *The Great Computer Language Shootout.*[1]

The great merit of functional languages is that they are easy to reason about. And reasoning is important. Take for instance *Proof Carrying Code* [74], where a server accepts an untrusted piece of code called an *agent*, only if the piece of code is accompanied by a proof that can be checked against certain safety policies set up by the server. Creating a proof for a particular piece of code cannot, in general, be done automatically, so the author of the code must provide the proof.

It is even easier to reason about non-optimised programs. Clearly, it is easier

[1] `http://www.bagley.org/~doug/shootout/`

to prove that

$$\begin{array}{lcl}
\mathit{quicksort}\ (\leq)\ [] & = & [] \\
\mathit{quicksort}\ (\leq)\ (x : xs) & = & \mathit{quicksort}\ (\leq)\ [y \mid y \leftarrow xs, \mathit{not}\ (x \leq y)] \\
 & & \mathbin{+\!\!+}\ [x] \\
 & & \mathbin{+\!\!+}\ \mathit{quicksort}\ (\leq)\ [y \mid y \leftarrow xs, x \leq y] \\
[]\ \mathbin{+\!\!+}\ ys & = & ys \\
(x : xs)\ \mathbin{+\!\!+}\ ys & = & x : (xs\ \mathbin{+\!\!+}\ ys)
\end{array}$$

is a stable sorting algorithm, than proving the same for the equivalent program[2]

$$\begin{array}{lcl}
\mathit{quicksort}\ (\leq)\ xs & = & \mathit{qs}\ (\leq)\ xs\ [] \\
\mathit{qs}\ (\leq)\ []\ y & = & y \\
\mathit{qs}\ (\leq)\ [a]\ y & = & a : y \\
\mathit{qs}\ (\leq)\ (a : x)\ y & = & \mathit{part}\ (\leq)\ a\ x\ []\ []\ y \\
\mathit{part}\ (\leq)\ a\ []\ l\ r\ y & = & \mathit{revqs}\ (\leq)\ l\ (a : \mathit{revqs}\ (\leq)\ r\ y) \\
\mathit{part}\ (\leq)\ a\ (b : x)\ l\ r\ y & = & \mathbf{if}\ a \leq b\ \mathbf{then}\ \mathit{part}\ (\leq)\ a\ x\ l\ (b : r)\ y \\
 & & \mathbf{else}\ \mathit{part}\ (\leq)\ a\ x\ (b : l)\ r\ y \\
\mathit{revqs}\ (\leq)\ []\ y & = & y \\
\mathit{revqs}\ (\leq)\ [a]\ y & = & a : y \\
\mathit{revqs}\ (\leq)\ (a : x)\ y & = & \mathit{revpart}\ (\leq)\ a\ x\ []\ []\ y \\
\mathit{revpart}\ (\leq)\ a\ []\ l\ r\ y & = & \mathit{qs}\ (\leq)\ l\ (a : \mathit{qs}\ (\leq)\ r\ y) \\
\mathit{revpart}\ (\leq)\ a\ (b : x)\ l\ r\ y & = & \mathbf{if}\ b \leq a\ \mathbf{then}\ \mathit{revpart}\ (\leq)\ a\ x\ (b : l)\ r\ y \\
 & & \mathbf{else}\ \mathit{revpart}\ (\leq)\ a\ x\ l\ (b : r)\ y\ \ .
\end{array}$$

Even more so if the $\mathbin{+\!\!+}$ operator is a library function that already has useful proofs associated with it, which leads us to the importance of modularity.

Nowadays, a programmer will not do very well in terms of productivity if he is not a good software engineer. An engineer knows how build or assemble things from smaller, well-defined components, and so-called component-based programming is the only way to avoid writing similar pieces of code over and over again. Of course, in terms of running-time and space consumption, the efficiency of a program assembled from independent components may very well be lower than an equivalent hand-crafted, hand-optimised program. But optimising by hand is both error prone and makes maintenance a nightmare.

In summary, there are several advantages from working with generic components programmed in a high-level language, but the main disadvantage is that

[2] Due to Lennart Augustsson.

one will inevitably pay a performance penalty if the individual components cannot be specialised relative to each other when fitted together.

1.1 Program Transformation

The above paragraph reveals that, when I speak of program transformation, I generally mean source-to-source code translation, not source-to-object code translation. Of course, the latter is possible and is the subject of research in *implementation of declarative languages.*

There exist numerous program-transformation techniques, but the techniques that will constitute the topic of this thesis are all based on so-called *driving*, or speculative execution, as it were. Since their purpose is to increase program speed, we may speak of *optimisation* rather than the more broad *transformation* (although I will give an example of the rather exotic technique called *program inversion* in Chapter 11). Thus, the field of investigation is really *optimisation of functional programs by driving-based transformation.*

1.1.1 Properties of Program Transformers

In a draft version of an invited paper for *Principles, Logics, and Implementations of High-level Languages 2002*, Jones and Glenstrup list the following desirable properties of a program generator.

1. High automation (user need not understand incomplete output code)
2. Successful generation process (no failures for well-formed input)
3. Efficient generation (fast enough)
4. Efficient generated program (fast enough)
5. Acceptable size of generated program (no size explosion)
6. Predictable results of generation (e.g., behaviour is not worsened)
7. Termination (the generator must terminate for all inputs)

In this thesis I will focus on items 1, 2, 4, 6 and 7. For any programmer to fully trust a program transformer which he is not familiar with, he needs to have certain guarantees. The most important guarantee is that the transformation never produces a program that is worse that the original. Given that I will work with non-strict languages, it is almost impossible to guarantee efficiency in absolute terms, but it should still be possible to guarantee that the produced programs behave as least as well *asymptotically*. If the programmer can trust the transformer in this respect, he can concentrate on the algorithmic aspects of the program, or, to put it a bit differently,

> "However, there is one aspect of functional programming that no amount of cleverness on the part of the compiler writer is likely to mitigate — the use of inferior or inappropriate data structures."
> *Chris Okasaki* [77]

I will further strengthen item 1 and require a transformer to be *fully automatic*, so that it can be used as a black box or as part of a compiler. That is, I shall study programs that work on programs, as depicted in Figure 1.1.[3] The former kind of transformer program may be part of an optimising compiler, but one could also imagine stand-alone transformers as well as transformers that are part of a programming environment.

Such uses of transformation techniques for functional programs (i.e., in optimising compilers, programming environments, etc.), have been popular in the academic society since the late 1970s, and the field is still very active with a conference devoted exclusively to the subject and a number of productive research groups around the world.

I will from time to time present small, seemingly uninteresting programs. The reason for using such contrived programs is twofold.

The first is simplicity: Realistic programs take up too much space, and usually the interesting points are obfuscated by uninteresting details. Furthermore, it is (almost) always the case that if there exists *one* contrived program that exhibits some particular property, there exist *infinitely many* programs that also exhibit this property. In any case, if I can contrive a program that raises a problem for a particular transformer, I cannot simply ignore the problem.

The second reason is immaturity: It is cumbersome to work with big programs, because the object language is too simple for real programming, and

[3] The idea of presenting program transformers as mincers is due to Anders Bondorf.

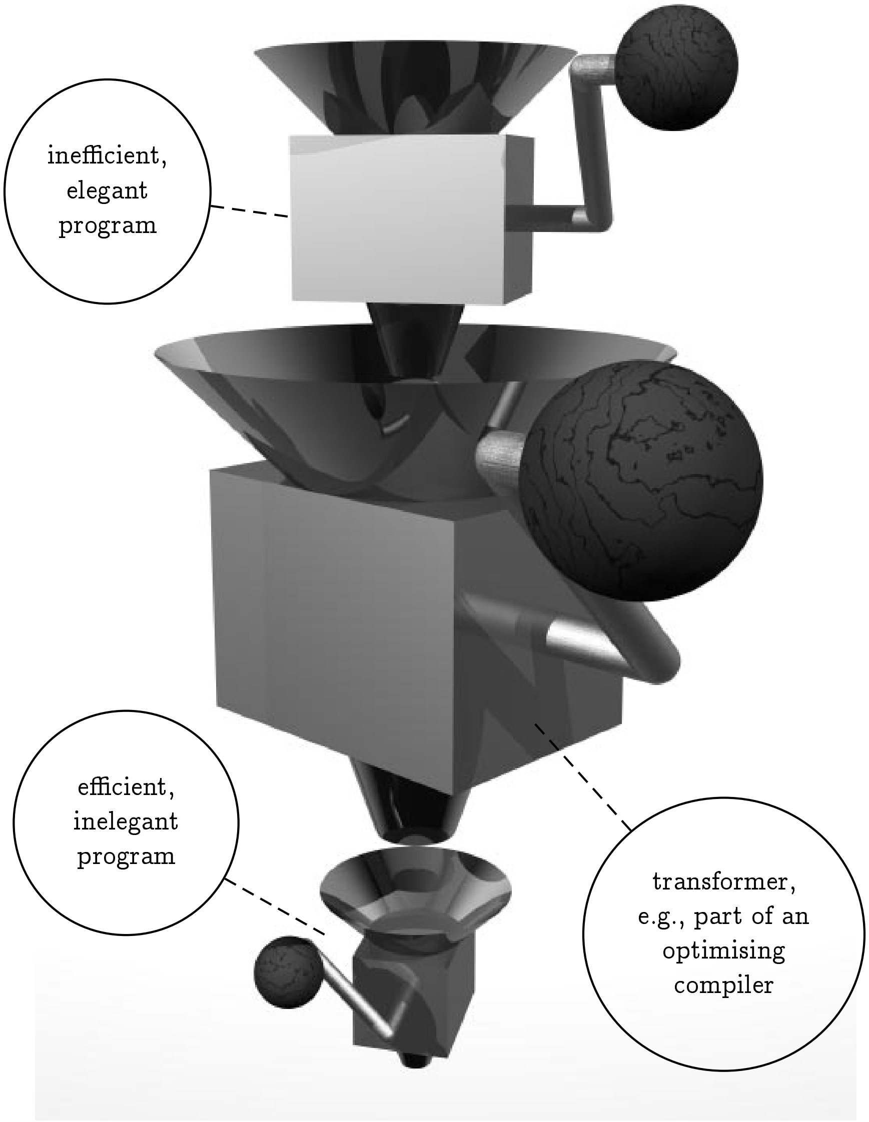


Figure 1.1: Automatic program transformation.

the techniques presented in this thesis exist in prototype implementations only, which makes it a somewhat painful experience to conduct experiments. Moreover, some of the techniques are very space and time consuming, contrary to items 3 and 5 of the preceding list, which means that some of the techniques do not really scale up. Further research is certainly needed in this area.

1.1.2 Example: Partial Evaluation

The most well-known automatic program-transformation technique is *partial evaluation*. The theoretical foundations of partial evaluation goes back to Kleene's S-M-N Theorem [54] which states, roughly, that there exists a *program specialiser* $\mathtt{mix}_n^m$ such that when $\mathtt{mix}_n^m$ is applied to a program q of m inputs and values v^n for the first n parameters of q (where $n \leq m$), then the result value of this application is a new program q' whose result on any $m - n$ values $v_{n+1} \ldots v_m$ is the same as the result value of q on v^m. In symbols

$$\begin{array}{ll} & \mathtt{mix}_n^m(q, v^n) = q' \\ \text{where} & \\ & q'(v_{n+1} \ldots v_m) = q(v^m) \ . \end{array} \tag{1.1}$$

One very simple way of constructing q' from q is simply to assign the values v^n to the first n parameters of q. For instance, let q be the following functional program which defines the function *pow* y x that computes the y'th power of x:

$$\begin{array}{lcl} pow\ 0\ x & = & 1 \\ pow\ (y+1)\ x & = & (pow\ y\ x) * x \ . \end{array}$$

With $m = 2$ and $n = 1$ (i.e., two arguments, the first known) and the value 3 for y I could construct the program q' of one argument by simply adding a new function

$$pow'\ x \quad = \quad pow\ 3\ x \ . \tag{1.2}$$

However, this is not what I have in mind. The point is that the new program q' should be *efficient*; all the computations inside q that depend only on the known input should be calculated once and for all. Thus, in the above example I expect a program q' like

$$pow'\ x \quad = \quad x * x * x \ .$$

You may have noted that specialisers are not transformers in the sense of Figure 1.1: A specialiser receives an input program *as well as part of the input* for this program. However, this is just a small variation of the theme. You can imagine that the specialiser as input is given the naïvely specialised program (i.e., the program where the first parameters are simply assigned the known values, as in Equation 1.2), and should produce as output the efficient specialised program.

Partial evaluation has been studied extensively in the academic society. One of the most interesting (and mind-boggling) aspects of this study is the *Futamura equations* which show that a specialiser, that is, a program $\texttt{mix}^{\mathtt{m}}_{\mathtt{n}}$ satisfying Equation 1.1 can be used to construct compilers (from interpreters) and compiler-generators. This has been known since Futamura's classical 1971 paper [31], but was first achieved in practice at University of Copenhagen in 1984 by Jones, Sestoft and Søndergaard [47], and the involved techniques have since been refined to produce compilers comparable to commercially available compilers in quality of compiled code.

The use of partial evaluation to speed up programs is now so well-studied that several research projects attempt to use the technique for industrial applications, and partial evaluation has been studied also for a number of non-declarative programming languages, e.g., C and Fortran.

1.1.3 Other Transformations

In this thesis I shall study a number of *other* program-transformation techniques than partial evaluation. The reason for studying these other techniques is partly because partial evaluation has already been studied intensively, but also because these other techniques are more aggressive than partial evaluation, in that they produce programs that are "more optimised" than those output by a typical partial evaluator. There is a price to be paid for the increased speedups: These transformers are more complicated and, presently, less well understood. The transformers I have in mind include (for functional languages) various kinds of *supercompilation*, *deforestation* and *tupling*, and (for logic languages) various kinds of *partial deduction*.

These can all be seen as instances of Burstall and Darlington's classical framework (from 1977) for transforming functional programs [13], which was also developed for logic programs by Tamaki and Sato [112]. Moreover, the different instances can be seen to have quite a lot in common. In fact, a main theme of the text will be to emphasise these similarities, even across language boundaries. More precisely, I shall present the above-mentioned transformation techniques in a uniform manner and show that many of the theoretical and practical problems pertaining to the transformers are *the same* for each transformer. Some of the most interesting of these problems is to make sure that the transformers terminate on all input programs, and to make sure that a transformed program is equivalent to the original program.

Another main theme will be to study and develop solutions to these problems, again emphasising similarities. A unified theory will be presented which aims to give a common foundation for studying and solving these problems for many transformers.

1.2 Contributions

The main contributions of this thesis are the following.

- I have improved existing driving-based program-transformation techniques by making them more powerful without sacrificing automation. I have achieved this by rephrasing driving-based program transformation in a setting where terms are represented as acyclic directed graphs (dags) to avoid duplication of computation. The dag setting has also prompted for solutions of how to ensure termination of the transformer, which has led to novel ideas for new *generalisation* operations. The dag-based term representation and the new generalisation operations will together perform caching of intermediate results and elimination of multiple traversals of data structures, in addition to elimination of intermediate data structures. The result is that transformed programs might be asymptotically faster than their original counterparts.

- This new setting has prompted for a better understanding of dag-based evaluation, which has led to the development of a *parameterised* operational semantics of orthogonal term-rewriting systems, in the sense that the underlying graph machinery can freely be chosen from within a wide range of possible implementation strategies without affecting the observable behaviour of programs.

- I have developed a notion of dag-based grammars with an expressiveness that lies on border between context-free and context-sensitive string grammars. The dag-based grammars can be used to give a concise representation of the input–output behaviour of programs, which I have illustrated by devising a method for automatic program inversion.

- I establish an interesting connection between program transformation of simple constraint checkers and binary-decision diagrams, which can be used in interactive product-line configuration.

1.3 Overview

This thesis is divided into two parts.

Part I: The first part presents various existing driving-based program-transformation techniques in a coherent form. All the transformations are expressed on the same programming language, namely a first-order, non-strict, functional language. This object language is described in terms of orthogonal term-rewrite systems, for which there exists a large body of off-the-shelf results. By presenting the various techniques in a such a unified framework, their similarities and differences should hopefully become clear. The main text of the chapters in the first part is kept free of references to the literature, but every chapter has a section that points out the origins of the material in that particular chapter. However, if concepts are fairly common, I instead refer to background material that surveys the field. The borderline between "established" knowledge and pristine discoveries is ever moving, so some people might not agree with my cut.

Part II: The second part presents some of the contributions I have made during the course of my Ph.D. education. All chapters in this part are centred around acyclic directed graphs (dags). Chapters 8 and 9 present a program-transformation framework based on dag rewriting, as opposed to the term rewriting approaches described in Part I. The graph-based framework makes it possible to achieve most of the effects from the existing driving-based techniques, as well as providing certain guarantees about the transformed programs and the transformation process itself. The framework also expresses program transformation in a more realistic setting, namely that of graph reduction, which is the standard way to implement non-strict functional languages. In Chapter 10

I describe an implementation of the dag-based transformer and show the results of transformation for various program. Chapter 11 describes how graph-based program-transformation can be used to invert programs into graph grammars that describe the set inputs giving a particular output. Chapter 12 shows how program transformation can be used for interactive product-line configuration, and an interesting relation to binary-decision diagrams is established. Part II is rounded up in Chapter 13 by a summary of the contributions, a critical assessment of the results, and future directions for further work in this area.

Part I

Introduction to Term-based Transformations

Chapter 2

Unfold–fold Transformation

In this chapter I present the fundamentals of Burstall and Darlington's classical "Transformation system for developing recursive programs", often called the *unfold–fold framework*. I will not only study the operations used to transform programs, but also discuss various strategies for and effects of applying these operations. Some preliminary remarks are also made regarding the problems of *correctness*, *completeness*, *efficiency* and *automation* which will be the focal points in this text. Be aware, however, that in this chapter I will be very informal about the semantics of the programming language used in the examples. I appeal to the readers intuition and postpone a precise definition of such matters to Chapter 3.

2.1 Unfolding and Laws

Consider the following functional program.

$$\begin{array}{lll}
(1) & \mathit{pow}\ 0\ x & = 1 \\
(2) & \mathit{pow}\ (y+1)\ x & = (\mathit{pow}\ y\ x) * x \\
(3) & \mathit{cube}\ x & = \mathit{pow}\ 3\ x\ \ .
\end{array}$$

Here $\mathit{pow}\ y\ x$ computes the y'th power of x. Therefore, $\mathit{cube}\ x$ computes the cube of x. definition 3 uses a general function, namely pow, to solve a specific problem, namely computation of cubes. This is in accordance with good programming style: We may face several specific problems, and by writing a single general function to solve all the special cases, we obtain a program which is simple and easy to maintain, contrary to a program consisting of a collection of specialised functions. For instance, a shift in data representation might cause a single change in the general program, but one change for each function in the specialised program.

By using definition 3, I have adopted a programming principle one might call *program specialisation*. As mentioned in the preface, the principle leads to elegant and highly maintainable programs, but not necessarily to efficient programs. However, with a few simple transformations I can obtain a better program. Consider again the definition 3:

(3) $\mathit{cube}\ x = \mathit{pow}\ 3\ x$.

How does one compute $\mathit{pow}\ 3\ x$? This must necessarily be done by using definition 2. In fact, I could use definition 2 to replace $\mathit{pow}\ 3\ x$ by $(\mathit{pow}\ 2\ x) * x$, thus obtaining the alternative definition

(4) $\mathit{cube}\ x = (\mathit{pow}\ 2\ x) * x$.

The step from definition 3 to definition 4 is called an *unfold step*, because I unfold the definition of the function pow. Another unfold step yields

(5) $\mathit{cube}\ x = (\mathit{pow}\ 1\ x) * x * x$.

Yet another unfold step gives me

(6) $\mathit{cube}\ x = (\mathit{pow}\ 0\ x) * x * x * x$.

A last unfold step gives me

(7) $\mathit{cube}\ x = 1 * x * x * x$.

Since there is nothing further to unfold, I may clean up the last definition by using the *algebraic law* which states that $1 * x = x$ to arrive at the definition

(8) $\mathit{cube}\ x = x * x * x$.

By replacing definition 3 in the original program with definition 8, I get the following program:

(1) $\mathit{pow}\ 0\ x \quad\quad = 1$
(2) $\mathit{pow}\ (y+1)\ x = (\mathit{pow}\ y\ x) * x$
(8) $\mathit{cube}\ x \quad\quad\; = x * x * x$.

Any term[1] I could evaluate in the previous original program I can also evaluate in this new program, and vice versa: The new program defines the same functions as the original one. However, the new definition of *cube* is more efficient than the original one since the unfold steps have been performed once and for all. As far as computing cubes is concerned, I could do with the specialised definition 8 alone; definition 1 and definition 2 are no longer needed.

The rôle of a program specialiser is to perform *efficient program specialisation*. With a program specialiser, one gets the best of both worlds: one can write elegant general programs and automatically get specialised efficient programs. Most people would agree that in a program using squares, cubes, etc., one should write a single power function and use it for the various special cases. The partial evaluator then generates the specialised versions automatically. The idea is that one *maintains the general version* of the program, and *uses the partial evaluator as part of the compilation process*. The benefit is clear: If one later finds a more clever way to calculate the power function, for instance

$$\begin{array}{lll}
(9) & \mathit{pow}\ 0\ x & = 1 \\
(10) & \mathit{pow}\ y\ x & = \mathbf{if}\ (y\ \mathbf{mod}\ 2) == 0\ \mathbf{then}\ \mathit{pow}\ (y/2)\ (x * x) \\
 & & \ \mathbf{else}\ x * \mathit{pow}\ (y-1)\ x \quad ,
\end{array}$$

such definitions could replace the definition of *pow* in the original program and the partial evaluator would take care of the specialisation. Whether the alternative formulation is better than the original depends on the implementation of the language.

2.2 Folding and Instantiation

Consider again the power program, but now suppose the base rather than the exponent is known.

$$\begin{array}{lll}
(1) & \mathit{pow}\ 0\ x & = 1 \\
(2) & \mathit{pow}\ (y+1)\ x & = (\mathit{pow}\ y\ x) * x \\
(11) & \mathit{ebuc}\ y & = \mathit{pow}\ y\ 3 \quad .
\end{array}$$

Again, I hope to transform the program into a more efficient one in which computations involving the known constant 3 are performed once and for all. How

[1] I will use "term" and "expression" interchangeably throughout this text.

is *pow* y 3 computed? Well, that depends on y. If y is 0, I will use definition 1, and otherwise I will use definition 2. I can take this into account by an *instantiation step* which gives from definition 11 the following two alternative definitions.

$$\begin{array}{lll} (12) & \mathit{ebuc}\ 0 & = \mathit{pow}\ 0\ 3 \\ (13) & \mathit{ebuc}\ (y+1) & = \mathit{pow}\ (y+1)\ 3\ \ . \end{array}$$

The reason that this is called an instantiation step is that I transform a definition into a number of *instances*. An instance of a term, definition, etc. is what is obtained by replacing one or more variables $x_1, x_2, \ldots, x_n$ with corresponding terms $t_1, t_2, \ldots, t_n$. All occurrences of the same variable x_i must be replaced by the same term t_i, and thus definition 12 is the instance of definition 11 obtained by replacing y by 0 and definition 13 is another instance of definition 11 obtained by replacing y by $y + 1$.

After the instantiation, I can perform unfold steps in definition 12 and definition 13, yielding

$$\begin{array}{lll} (14) & \mathit{ebuc}\ 0 & = 1 \\ (15) & \mathit{ebuc}\ (y+1) & = (\mathit{pow}\ y\ 3) * 3\ \ . \end{array}$$

How can I proceed with this definition? The most complicated part of the definition is the call *pow* y 3. But the computation of *pow* y 3 is exactly what I wanted *ebuc* y to do, as can be seen from definition 11. I therefore replace *pow* y 3 by *ebuc* y yielding

$$\begin{array}{lll} (16) & \mathit{ebuc}\ (y+1) & = (\mathit{ebuc}\ y) * 3\ \ . \end{array}$$

Such a step is called a *fold step*, since it does the opposite of an unfold step. Whereas an unfold step replaces a term that "matches" the left-hand side of a definition by the corresponding right-hand side, a fold step replaces a term that "matches" the right-hand side of a definition by the corresponding left-hand side.

By taking definition 1, definition 2, definition 14 and definition 16, I obtain the following program.

$$\begin{array}{lll} (1) & \mathit{pow}\ 0\ x & = 1 \\ (2) & \mathit{pow}\ (y+1)\ x & = (\mathit{pow}\ y\ x) * x \\ (14) & \mathit{ebuc}\ 0 & = 1 \\ (16) & \mathit{ebuc}\ (y+1) & = (\mathit{ebuc}\ y) * 3\ \ . \end{array}$$

It is worth noting that I have managed to introduce a new *recursive* function $ebuc$ by means of the fold step. Again, if I am only interested in computing 3^y for various y, I can do with definition 14 and definition 16 alone. It should be noted that the new version of $ebuc$ is hardly more *efficient* than the original version, but its definition has been made more transparent and direct.

2.3 Abstraction and Definition

As indicated in the two examples above, transformation in the unfold–fold framework starts out from some program with a number of definitions, applies a number of elementary operations yielding more definitions, and finally selects a set of definitions to constitute the new program. These elementary operations include:

1. Unfolding.
2. Folding.
3. Instantiation.
4. Laws.

There are two more operations which have not yet been introduced, namely

5. Abstraction.
6. Definition.

Abstraction steps replace several identical subexpressions by a let-bound variable as in the step from

$$f\ x = (g\ x) + (g\ x)$$

to

$$f\ x = \mathbf{let}\ y := g\ x\ \mathbf{in}\ y + y\ \ .$$

Such a step will be useful for advanced forms of common subexpression elimination, known as *tupling.*

Definition steps introduce fresh functions, and this will be useful for enabling folding steps, as I shall see later.

2.4 Strategies

The operations presented so far, that is, instantiation, unfolding, etc., are just the basic building blocks used for program transformation. They do not constitute an algorithm because for any given program there may be numerous different ways to apply the operations: Which definition to consider next, which call in a definition to consider, which operation to apply to the call, etc.

In the quest for full automation of program transformation, I can impose what is called a *strategy* on the use of the above operation. Which strategy I use depends on what I try to achieve by the transformation. For instance, a partial evaluator may be seen as a strategy aiming at efficient program specialisation. Later I shall study several other strategies in detail, including

deforestation aiming at program composition.

tupling aiming at a form of common subexpression elimination.

supercompilation aiming at several different effects.

In the following, I will give examples illustrating some of these strategies. Be aware, however, that a strategy not necessarily constitute a concrete *algorithm*, in the sense that an algorithm must use these operations in some *deterministic* way.

In the rest of this thesis, I will use the usual notation for lists when convenient: $[]$ means the empty list, and $x : xs$ means the list consisting of the head x and the tail xs. I also write $[\,t_1, \ldots, t_n\,]$ to denote lists of length n and I will use the convention that function application binds stronger than any infix constructor.

2.4.1 Example of deforestation

Consider the following program for appending lists.

$$\begin{array}{lll} (17) & \mathit{append}\ []\ \mathit{ys} & = \mathit{ys} \\ (18) & \mathit{append}\ (x : \mathit{xs})\ \mathit{ys} & = x : \mathit{append}\ \mathit{xs}\ \mathit{ys} \\ (19) & \mathit{dappend}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} & = \mathit{append}\ (\mathit{append}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}\ \ . \end{array}$$

The function *dappend* appends three lists by calling *append* twice. The definition of *dappend* is simple and easy to understand, but inefficient since *xs* is traversed twice.

Analogously to the case of program specialisation, the problem is that I have composed the calls to *append* in a naïve way. One might say that the rôle of *deforestation* is to perform efficient *program composition*, as I now illustrate. By instantiation of definition 19:

$$\begin{array}{lll}(20) & \mathit{dappend}\ []\ \mathit{ys}\ \mathit{zs} & = \mathit{append}\ (\mathit{append}\ []\ \mathit{ys})\ \mathit{zs} \\ (21) & \mathit{dappend}\ (x : \mathit{xs})\ \mathit{ys}\ \mathit{zs} & = \mathit{append}\ (\mathit{append}\ (x : \mathit{xs})\ \mathit{ys})\ \mathit{zs}\ \ .\end{array}$$

By unfolding the inner call to *append* in definition 20:

$$(22)\ \ \mathit{dappend}\ []\ \mathit{ys}\ \mathit{zs} = \mathit{append}\ \mathit{ys}\ \mathit{zs}\ \ .$$

By unfolding the inner call to *append* in definition 21:

$$(23)\ \ \mathit{dappend}\ (x : \mathit{xs})\ \mathit{ys}\ \mathit{zs} = \mathit{append}\ (x : \mathit{append}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}\ \ .$$

Another unfold step of the outer call to *append*:

$$(24)\ \ \mathit{dappend}\ (x : \mathit{xs})\ \mathit{ys}\ \mathit{zs} = x : \mathit{append}\ (\mathit{append}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}\ \ .$$

And finally, by a fold step:

$$(25)\ \ \mathit{dappend}\ (x : \mathit{xs})\ \mathit{ys}\ \mathit{zs} = x : \mathit{dappend}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs}\ \ .$$

Thus, definition 17, definition 18, definition 22 and definition 25 give

$$\begin{array}{lll}(17) & \mathit{append}\ []\ \mathit{ys} & = \mathit{ys} \\ (18) & \mathit{append}\ (x : \mathit{xs})\ \mathit{ys} & = x : \mathit{append}\ \mathit{xs}\ \mathit{ys} \\ (22) & \mathit{dappend}\ []\ \mathit{ys}\ \mathit{zs} & = \mathit{append}\ \mathit{ys}\ \mathit{zs} \\ (25) & \mathit{dappend}\ (x : \mathit{xs})\ \mathit{ys}\ \mathit{zs} & = x : \mathit{dappend}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs}\ \ .\end{array}$$

This program is more efficient since *dappend* now traverses its first parameter only once. Another way to view the optimisation is that the *intermediate* list built by the inner call to *append* in definition 19 is no longer constructed in definition 22 or definition 25. Hence, deforestation is said to *eliminate intermediate data structures*.

A suitable strategy for elimination of intermediate data structures could be the following.

1. If it is impossible to unfold, either stop transformation or continue.

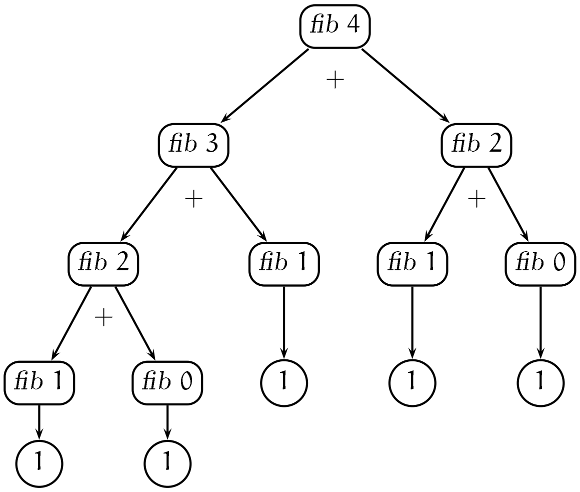


Figure 2.1: Computation of the fourth Fibonacci number.

2. Instantiate.
3. Unfold calls repeatedly.
4. Fold where possible.
5. Repeat step 1.

2.4.2 Example of tupling

I will now present a more radical example of an optimisation. Consider the following definition of the Fibonacci function.

$$\begin{array}{lll} (26) & \mathit{fib}\ 0 & = 1 \\ (27) & \mathit{fib}\ 1 & = 1 \\ (28) & \mathit{fib}\ (x+2) & = \mathit{fib}\ (x+1) + \mathit{fib}\ x\ \ . \end{array}$$

It is not hard to see that this definition requires exponential time in x to compute *fib* x. For instance, *fib* 4 gives rise to the call tree in Figure 2.1.

In general, the rightmost branch in the call tree for *fib* x is shorter than every other branch, and the rightmost branch has $\lfloor \frac{x}{2} \rfloor + 1$ edges, so the whole tree has at least

$$2^{\lfloor \frac{x}{2} \rfloor} \geq 2^{\frac{x}{2}-1} = \frac{2^{\frac{x}{2}}}{2} = \frac{{2^{\frac{1}{2}}}^{x}}{2} = \frac{1}{2}\left(\sqrt{2}\right)^{x}$$

nodes of form *fib* y, so evaluation of *fib* x takes exponential time.

It is easy to identify the source of the exponential running time: The same calls are evaluated over and over again. For instance, in Figure 2.1 the call to *fib* 3 leads to a call to *fib* 2 although this call is also made by the parent *fib* 4; also, *fib* 3 calls *fib* 1 although *fib* 2 also does this.

The idea of tupling is to eliminate these multiple computations of the same value. I can transform definition 28 into

(29) $\mathit{fib}\ (x+2) = \mathbf{let}\ \langle y\ ,\ z\rangle := \langle \mathit{fib}\ (x+1)\ ,\ \mathit{fib}\ x\rangle\ \mathbf{in}\ y+z$.

Note that this is (a variant of) what I have called an abstraction step. By a *definition step*, I can introduce a new function g defined by

(30) $g\ x = \langle \mathit{fib}\ (x+1)\ ,\ \mathit{fib}\ x\rangle$.

By a fold step in definition 29, I get

(31) $\mathit{fib}\ (x+2) = \mathbf{let}\ \langle y\ ,\ z\rangle := g\ x\ \mathbf{in}\ y+z$.

Instantiation of definition 30:

(32) $g\ 0 \qquad\quad = \langle \mathit{fib}\ 1\ ,\ \mathit{fib}\ 0\rangle$
(33) $g\ (x+1) = \langle \mathit{fib}\ (x+2)\ ,\ \mathit{fib}\ (x+1)\rangle$.

Unfolding definition 32 twice:

(34) $g\ 0 = \langle 1\ ,\ 1\rangle$.

Unfolding definition 33:

(35) $g\ (x+1) = \langle \mathit{fib}\ (x+1) + \mathit{fib}\ x\ ,\ \mathit{fib}\ (x+1)\rangle$.

Abstraction of definition 35:

(36) $g\ (x+1) = \mathbf{let}\ \langle y\ ,\ z\rangle := \langle \mathit{fib}\ (x+1)\ ,\ \mathit{fib}\ x\rangle\ \mathbf{in}\ \langle y+z\ ,\ y\rangle$.

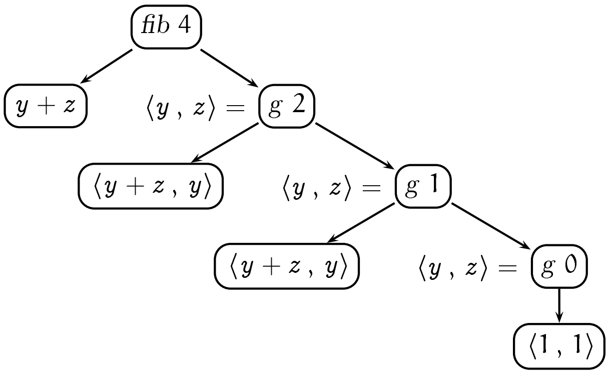


Figure 2.2: Improved computation of the fourth Fibonacci number.

Folding definition 36 with definition 30 gives

$$(37)\quad g\ (x+1) = \mathbf{let}\ \langle y\ ,\ z\rangle := g\ x\ \mathbf{in}\ \langle y+z\ ,\ y\rangle \quad .$$

The final program now consists of

$$\begin{array}{lll}
(26) & \mathit{fib}\ 0 & = 1 \\
(27) & \mathit{fib}\ 1 & = 1 \\
(31) & \mathit{fib}\ (x+2) & = \mathbf{let}\ \langle y\ ,\ z\rangle := g\ x\ \mathbf{in}\ y+z \\
(34) & g\ 0 & = \langle 1\ ,\ 1\rangle \\
(37) & g\ (x+1) & = \mathbf{let}\ \langle y\ ,\ z\rangle := g\ x\ \mathbf{in}\ \langle y+z\ ,\ y\rangle \quad .
\end{array}$$

As Figure 2.2 illustrates, the call tree and thereby the running time of *fib* x is now linear in x.

A suitable strategy for tupling could be the following:

1. If it is impossible to unfold, either stop transformation or continue.
2. Abstract calculations that depend on the same variable into a tuple.
3. Introduce new functions calculating the tuples introduced.
4. If it is impossible to unfold, instantiate.
5. Unfold calls repeatedly.
6. Fold where possible.
7. Repeat step 1.

2.4.3 Other strategies and rules

Of course, there are plenty of other conceivable strategies than those mentioned above. For instance, Pettorossi [80] is able to transform the Fibonacci program into a program which runs in $\mathrm{O}(\log \mathrm{n})$. There is also another rule appearing in Burstall and Darlington's paper, namely *redefinition*. Suppose you are given a number of definitions $\mathrm{d}_1, \ldots \mathrm{d}_\mathrm{n}$ and suppose you wish to replace these by some definition d. Then this is allowed, provided you can transform d into $\mathrm{d}_1, \ldots \mathrm{d}_\mathrm{n}$. For instance, given

(38) $f\ 0 \qquad = 0$
(39) $f\ (x+1) = f\ x$,

you cannot get the definition

(40) $f\ x = 0$

using the previous six rules. However, definition 40 can be transformed into definition 38 and definition 39, so the latter may be replaced by the former. Such a step, however, changes the semantics — from non-strict to strict evaluation — an issue I will not discuss further in this chapter.

2.5 Correctness, Completeness, Automation

Having transformed some function f to a (hopefully more efficient) function f', the following issues should be addressed:

- Are f and f' *equivalent*? In particular, do f and f' *terminate* on the same inputs? Note that these questions only make sense relative to a *semantics* of the programming language in question. Problems of this sort is collectively referred to as the *correctness* of the transformation. *Completeness* concerns the opposite question: Given equivalent functions f and f', can f be transformed to f' by means of the rules? Although important in general, this question is trivial in the context of the given strategies, where incompleteness is always obvious.

- Is f' at least as efficient in terms of *time usage* as f? Of course, the question makes sense relative to a model of computation time only. The analogous question for *space usage* might also be asked, but seldom is, with the exception of the work carried out by Gustavsson and Sand [39].

- Is f' at most as *big* as f? Although program size in principle does not affect the running time, there may be a correlation in practice, and for this reason, among others, program size may also be an issue.

As mentioned earlier, I am interested in *algorithms* that perform optimisations. To what extent can the above examples of transformation be automated? As indicated, different strategies, that is, transformation algorithms, exist, and for each such strategy the most important problems are (apart from the problems mentioned above):

- Is termination guaranteed on all input programs? If not, then for which programs?

- If the transformation aims at eliminating certain forms of inefficiency, can one give any guarantees about what will be removed?

2.6 Further Reading

Burstall and Darlington's 1977 paper [13] is still very readable. More recent accounts of the classical unfold–fold framework covering both functional and logic languages, and which has more about strategies, is given by Pettorossi and Proietti [83, 84]. The latter two papers have many further references, for instance Baur and Wossner [6] and Partsch [78].

Partial evaluation is undoubtedly the program transformation technique that has been most intensively studied. A short introduction is given by Consel and Danvy [22], a longer one is given by Jones [49]. For a thorough treatment of partial evaluation, the book by Jones, Gomard and Sestoft [50] is a must.

The following chapters will introduce deforestation (Chapter 4), supercompilation (Chapters 5 & 6) and tupling (Chapter 7) in more detail.

Chapter 3

Functional programs

As illustrated in the previous chapter, certain transformations on a program can make the program more effective, but certain other transformations can make the program *less* effective. Even worse, some transformations can turn a perfectly good program into a non-terminating program, or a program that gives wrong results. To avoid such non-beneficial transformations, I need to have some way to relate programs w.r.t. semantics (i.e., meaning) and running time. The point of this chapter is therefore to pin down the semantics of programs. Throughout this thesis I will use a first-order, non-strict, functional language as the object of program transformation.

3.1 Syntax

You should be warned that I use a fair amount of non-standard notation. It might therefore be a good idea to browse through the sections of Appendix A.

Definition 1 Assume denumerable, pairwise disjoint sets of symbols referred to as *Constructor*, *Function*, *Matcher*, and *Variable*. All symbols have fixed arity, and variables in particular have arity 0. Then the set of programs and terms are defined by the abstract syntax grammar in Figure 3.1. I require that

1. No function or matcher name is defined more than once.

2. No two patterns in a matcher definition contain the same constructor.

$$x \in \mathit{Variable}, c \in \mathit{Constructor}, f \in \mathit{Function}, g \in \mathit{Matcher}, n \geq 0, m > 0$$

$$\begin{array}{rcccll}
\mathit{Program} & \ni & q & ::= & d^m & \text{(definitions)} \\
 & & d & ::= & f\ x^n = t & \text{(function)} \\
 & & & | & g\ p_1\ x^n = t_1 & \text{(matcher)} \\
 & & & & \vdots & \\
 & & & & g\ p_m\ x^n = t_m & \\
\mathit{Pattern} & \ni & p & ::= & c\ x^n & \text{(flat pattern)} \\
\mathit{Term} & \ni & t & ::= & x & \text{(variable)} \\
 & & & | & c\ t^n & \text{(construction)} \\
 & & & | & f\ t^n & \text{(application)} \\
 & & & | & g\ t^m & \text{(match)} \\
\mathit{Value} & \ni & v & ::= & c\ v^n &
\end{array}$$

Figure 3.1: Abstract grammar for the object language.

3. No variable occurs more than once in the left-hand side of a function definition (the definition is *left-linear*).

4. All variables in the body of a function or matcher definition are present among the variables in the left-hand side of the definition.

I let $f\ x^n \overset{q}{=} t$ denote that the program q contains a definition $f\ x^n = t$, and similarly for $g\ p\ x^n \overset{q}{=} t$. As a shorthand, I let

$$\mathit{Symbol} = \mathit{Constructor} \cup \mathit{Function} \cup \mathit{Matcher}\ .$$

The sequence resulting from enumerating the variables in a term t by their first occurrence, from left to right, is denoted $\mathscr{V}t$. A term t is a *ground* term iff $\mathscr{V}t = \langle\rangle$. □

Example 1 Assuming f is a function, CONS is a constructor, and everything else is a variable, $\mathscr{V}(\text{CONS}\ x\ (f\ y\ x)) = \langle x\ ,\ y\rangle \neq \langle y\ ,\ x\rangle = \mathscr{V}(f\ y\ (\text{CONS}\ y\ x))$.
□

Example 2 A program for appending lists could be written

$$\begin{array}{ll} \textbf{data}\ \text{List}\ a & = \text{NIL} \mid \text{CONS}\ a\ (\text{List}\ a) \\ \mathit{append}\ \text{NIL}\ \mathit{ys} & = \mathit{ys} \\ \mathit{append}\ (\text{CONS}\ x\ \mathit{xs})\ \mathit{ys} & = \text{CONS}\ x\ (\mathit{append}\ \mathit{xs}\ \mathit{ys})\ . \end{array}$$

□

Remark 1 The above program contains a data-type definition. For clarity, I will put such data-type definitions in my example programs, even though such data-type definitions are not permitted in the language. □

The preceding definition of the functional programming language is formulated so that it corresponds to the well-known class of *left-normal orthogonal term-rewrite system*[1] (see Klop [55] for an overview and references). The choice of term-rewrite systems as the foundation of the programming language is motivated by the substantial body of theory developed for such systems, and that it provides a great deal of freedom in defining the *semantics* of programs. In contrast to the usual practice, I will not impose a notion of type-correctness on programs because such correctness is not essential to the program-transformation techniques presented in the following chapters.

3.2 Semantics

The semantics of the language is based on rewriting terms that match definitions in the program.

Definition 2 Let $\mathsf{q} \in \mathit{Program}$ and $\{\mathsf{r}, \mathsf{t}, \mathsf{u}\} \subseteq \mathit{Term}$.

1. r is a q-*redex* iff q contains a definition $\mathsf{t} \overset{\mathsf{q}}{=} \mathsf{u}$ and there is a $\theta \in$ *Substitution* such that $\mathsf{r} = \mathsf{t}\theta$. If so, the term $\mathsf{u}\theta$ is called the *contractum* of r.
2. The relation $\xrightarrow[\mathsf{redex(q)}]{} \subseteq \mathit{Term} \times \mathit{Term}$, relates redexes to contractums: $\mathsf{t} \xrightarrow[\mathsf{redex(q)}]{} \mathsf{u}$ iff t is a q-redex and u is its contractum.

[1] Orthogonal means left-linear and non-ambiguous, that is, variables occur only once in the left-hand side of rewrite rule, and there is always exactly one rewrite rule that can rewrite a particular redex. O'Donnell [76] calls an orthogonal TRS *left-normal* if, for every rewrite rule $\mathsf{l} \to \mathsf{r}$, the constant and function symbols are to the left of the variables in l.

□

Definition 3 A $q \in \mathit{Program}$ induces a *rewrite* relation $\xrightarrow[\mathsf{trs}(q)]{} \subseteq \mathit{Term} \times \mathit{Term}$ as follows. Let $\{t, u, s\} \subseteq \mathit{Term}$ such that $t \xrightarrow[\mathsf{redex}(q)]{} u$ and $x \in \mathscr{V}s$. Then $s[x:=t] \xrightarrow[\mathsf{trs}(q)]{} s[x:=u]$. □

Example 3 Consider the list-append program from Example 2 and the datatype

data Colour = RED | GREEN | BLUE .

The term '*append*(CONS GREEN(CONS RED NIL))(CONS BLUE NIL)' can be rewritten as follows.

$$\begin{aligned} & \mathit{append}(\textsc{cons green}(\textsc{cons red nil}))(\textsc{cons blue nil}) \\ \xrightarrow[\mathsf{trs}(q)]{} & \textsc{cons green}(\mathit{append}(\textsc{cons red nil})(\textsc{cons blue nil})) \\ \xrightarrow[\mathsf{trs}(q)]{} & \textsc{cons green}(\textsc{cons red}(\mathit{append}\ \textsc{nil}(\textsc{cons blue nil}))) \\ \xrightarrow[\mathsf{trs}(q)]{} & \textsc{cons green}(\textsc{cons red}(\textsc{cons blue nil})) \end{aligned}$$

□

The point of term rewriting is usually to rewrite a ground term until a *normal form* is reached, that is, to reach a state where no more rewrites can take place, like in the example above.

Definition 4 A q-*normal-form* (shortened q-nf) is a term t for which there does not exists a term u such that $t \xrightarrow[\mathsf{trs}(q)]{} u$. □

A very important property of left-normal orthogonal rewrite systems is that, in some sense, it does not matter in which order one chooses to reduce the redexes: $\xrightarrow[\mathsf{trs}(q)]{}$ is *confluent*, meaning that it will always be possible to reach a unique normal (if one exists), no matter which rewrites have been performed. In other words, for any legal program q, any reduction sequence that reaches a normal form will reach the same normal as any other reduction sequence.

Definition 5 A relation $\rightarrow \subseteq \mathit{Term} \times \mathit{Term}$ is *confluent* iff

$$\forall \{t, t', t''\} \subseteq \mathit{Term} \left(t' \xleftarrow{*} t \xrightarrow{*} t'' \Rightarrow \exists u \in \mathit{Term} \left(t' \xrightarrow{*} u \xleftarrow{*} t'' \right) \right) .$$

□

It is customary to present propositions like the preceding one as diagrams where dashed arrows represents existential rewrites depending on universal rewrites, shown as full arrows. For instance, the displayed proposition in Definition 5 could be depicted

$$\begin{tikzcd}
t \arrow[r, "*"] \arrow[d, "*"] & t' \arrow[d, dashed, "*"] \\
t'' \arrow[r, dashed, "*"] & u
\end{tikzcd}$$

which provides a somewhat more intuitive understanding.

Lemma 1 $\xrightarrow[\mathsf{trs}(q)]{}$ *is confluent.*

PROOF. A consequence of the fact that every orthogonal term-rewrite system is confluent. See Klop [55] for a proof.

3.2.1 Correct programs

A ground term may fail to evaluate to a *value* for two reasons: Either the computation consists of an infinite number of steps, or the computation "gets stuck" at some point. The former reason is usually called *non-termination* and is an inherent unpleasantness in any universal programming language.

One can, however, circumvent most stuck computations by imposing a standard Milner–Hindley–Mycroft *polymorphic type system* [71, 42, 73] on the language to reject program–term pairs that will get stuck in a normal form which is not a value. Although the optimisations performed in state-of-the-art compilers for functional languages are often guided by type information in the intermediate language, such type information is not essential, or even needed, for most of the techniques presented in this thesis. I will therefore assume that standard compiler phases have discarded all programs with type-related errors, undefined functions, wrong arity, etc. I thus only consider programs q and terms t_0 for which any rewrite sequence

$$t_0 \xrightarrow[\mathsf{trs}(q)]{} t_1 \xrightarrow[\mathsf{trs}(q)]{} t_2 \xrightarrow[\mathsf{trs}(q)]{} \cdots$$

either

1. results in a *value* v, or
2. is infinite, or
3. results in a stuck application of a pattern-matching function to a constructor for which there is no matching pattern.

When limited amounts of type information is needed (cf. Chapter 6), I will simply assume that programs and terms are type annotated and thus type correct.

Example 4 Consider again the program q from Example 2. The term

$$\mathit{append}\ (\textsc{cons}\ 1\ \textsc{nil})\ (\textsc{cons}\ 2\ \textsc{nil})$$

can be rewritten to the value

$$\textsc{cons}\ 1\ (\textsc{cons}\ 2\ \textsc{nil})\ \ .$$

The term

$$\mathit{append}\ (\textsc{cons}\ 1\ \textsc{leaf})\ (\textsc{cons}\ 2\ \textsc{nil})$$

is not correct w.r.t. q, since rewriting gets stuck in the term

$$\textsc{cons}\ 1\ (\mathit{append}\ \textsc{leaf}\ (\textsc{cons}\ 2\ \textsc{nil}))\ \ .$$

□

Usually, one is interested in terminating programs only. But during transformation of programs, I will have to deal with program fragments and composition of program fragments. Such program fragments might not be terminating, even though the larger program they are part of is.

3.2.2 Non-termination and Observability

To account for non-terminating programs, I will define the semantics of a program as its observable behaviour. I will denote both unobservable behaviours and stuck computations by $\perp$, and I will extend the set of values to allow values containing $\perp$. After the example below, a precise definition follows.

Example 5 Consider the program

$$\begin{array}{ll} \textbf{data}\ \mathrm{Nat} & = \mathsf{0} \mid \textsc{succ}\ \mathrm{Nat} \\ \textbf{data}\ \mathrm{List}\ a & = \textsc{nil} \mid \textsc{cons}\ a\ (\mathrm{List}\ a) \\ \mathit{main}\ n & = \textsc{cons}\ n\ (\mathit{main}\ (\textsc{succ}\ n))\ . \end{array}$$

The term '*main* 0' will produce an infinite rewrite sequence

$$\begin{array}{ll} \mathit{main}\ \mathsf{0} \xrightarrow[\mathsf{trs(q)}]{} & \textsc{cons}\ \mathsf{0}\ (\mathit{main}\ (\textsc{succ}\ \mathsf{0})) \\ \xrightarrow[\mathsf{trs(q)}]{} & \textsc{cons}\ \mathsf{0}\ (\textsc{cons}\ (\textsc{succ}\ \mathsf{0})\ (\mathit{main}\ (\textsc{succ}\ (\textsc{succ}\ \mathsf{0})))) \\ \xrightarrow[\mathsf{trs(q)}]{} & \cdots \\ \vdots & \end{array}$$

I will say that the semantics of '*main* 0' w.r.t. the above program is the set

$$\left\{ \begin{array}{l} \bot \\ \textsc{cons}\ \bot\ \bot \\ \textsc{cons}\ \mathsf{0}\ \bot \\ \textsc{cons}\ \mathsf{0}\ (\textsc{cons}\ \bot\ \bot) \\ \textsc{cons}\ \mathsf{0}\ (\textsc{cons}\ (\textsc{succ}\ \bot)\ \bot) \\ \textsc{cons}\ \mathsf{0}\ (\textsc{cons}\ (\textsc{succ}\ \mathsf{0})\ \bot) \\ \cdots \end{array} \right\}$$

which has the infinite limit

$$\textsc{cons}\ \mathsf{0}\ (\textsc{cons}\ (\textsc{succ}\ \mathsf{0})\ (\textsc{cons}\ (\textsc{succ}\ (\textsc{succ}\ \mathsf{0}))\ (\ldots)))\ .$$

I will say that the semantics of '*main* 0' w.r.t. the program

$$\begin{array}{ll} \textbf{data}\ \mathrm{Nat} & = \mathsf{0} \mid \textsc{succ}\ \mathrm{Nat} \\ \textbf{data}\ \mathrm{List}\ a & = \textsc{nil} \mid \textsc{cons}\ a\ (\mathrm{List}\ a) \\ \mathit{main}\ n & = \textsc{cons}\ n\ \mathit{inf} \\ \mathit{inf} & = \mathit{inf} \end{array}$$

is the set $\{\bot, \textsc{cons}\ \bot\ \bot, \textsc{cons}\ \mathsf{0}\ \bot\}$ which has the finite limit $\textsc{cons}\ \mathsf{0}\ \bot$. □

As you might have inferred from the preceding example, the outermost constructors of a term are observed from left to right. If a computation gets stuck or diverges in the middle of a term, constructors to the right of the stuck or diverging term are not observable. To define this notion of observable behaviour, it is convenient to generalise observability to *sequences* of terms, because sequences maintain left-to-right order.

Definition 6 Let $\perp$ be a special constructor symbol not equal to any other symbols used in programs. Given $\mathrm{t} \in \mathit{Term}$, I let the *observability of* t, denoted $\mathscr{O}\mathrm{t}$, by a set of terms (extended with the special symbol $\perp$) defined by

$$\begin{array}{rcl}
\mathscr{O}\mathrm{t} & \overset{\mathrm{def}}{=} & \mathscr{O}'\langle \mathrm{t} \rangle \\
\mathscr{O}'\langle\rangle & \overset{\mathrm{def}}{=} & \{\langle\rangle\} \\
\mathscr{O}'\langle \mathrm{t}_0\,,\, \mathrm{t}^n \rangle & \overset{\mathrm{def}}{=} & \{\langle \overbrace{\perp, \ldots, \perp}^{n+1} \rangle\} \\
 & & \cup \begin{cases} \{\langle \mathrm{c}\ v^m\,,\, w^n \rangle \mid \langle v^m\,,\, w^n \rangle \in \mathscr{O}'\langle \mathrm{u}^m\,,\, \mathrm{t}^n \rangle\}\,, & \\ & \text{if } \mathrm{t}_0 = \mathrm{c}\ \mathrm{u}^m \\ \varnothing, & \text{otherwise.} \end{cases}
\end{array}$$

□

The above definition is well-founded by the finite structure of terms: Even though all terms referred to in $\mathscr{O}'\langle \mathrm{u}^m\,,\, \mathrm{t}^n \rangle$ in the right-hand side of the last line are subterms of the term on the left-hand side $\mathscr{O}'\langle \mathrm{t}_0\,,\, \mathrm{t}^n \rangle$, either t_0 is a 0-ary constructor, and thus the number of elements in the sequence decreases, or t_0 is not a 0-ary constructor, and thus some of the elements are *proper* subterms.

Definition 7 Given $\mathrm{q} \in \mathit{Program}$ and $\mathrm{t} \in \mathit{Term}$, I write $[\![\mathrm{t}]\!]_{\mathrm{q}} \subseteq \mathit{Term}$ for the *semantics of* t *w.r.t.* q defined by

$$[\![\mathrm{t}]\!]_{\mathrm{q}} \overset{\mathrm{def}}{=} \bigcup_{\mathrm{t} \xrightarrow[\mathsf{trs(q)}]{*} \mathrm{t}'} \mathscr{O}\mathrm{t}'$$

□

The semantics of a term w.r.t. a program is thus the observability of all derivations of the term, which might seem like a mess at first, but the definition is carefully modelled so that rewriting is monotonic w.r.t. to observability, as expressed in the following lemma.

Lemma 2 $t \xrightarrow[\mathsf{trs(q)}]{} u \quad \Rightarrow \quad \mathscr{O}t \subseteq \mathscr{O}u$.

PROOF (sketch). By case analysis of the structure and position of the contracted redex. □

The monotonicity of the observable behaviour together with unique normal forms give the desirable property that rewrites preserve semantics.

Lemma 3 $\quad t \xrightarrow[\mathsf{trs(q)}]{} u \Rightarrow \llbracket t \rrbracket_q = \llbracket u \rrbracket_q$.

PROOF. Assume $t \xrightarrow[\mathsf{trs(q)}]{} u$.

1. $\llbracket t \rrbracket_q \supseteq \llbracket u \rrbracket_q$:

$$\begin{aligned}
&\llbracket t \rrbracket_q \\
&= \bigcup_{t \xrightarrow[\mathsf{trs(q)}]{*} t'} \mathscr{O}\langle t' \rangle && \text{[by Definition 7]} \\
&= \bigcup_{t \xrightarrow[\mathsf{trs(q)}]{*} t'} \mathscr{O}\langle t' \rangle \cup \bigcup_{u \xrightarrow[\mathsf{trs(q)}]{*} t'} \mathscr{O}\langle t' \rangle && [t \xrightarrow[\mathsf{trs(q)}]{} u] \\
&= \bigcup_{t \xrightarrow[\mathsf{trs(q)}]{*} t'} \mathscr{O}\langle t' \rangle \cup \llbracket u \rrbracket_q && \text{[by Definition 7]} \\
&\supseteq \llbracket u \rrbracket_q \ . && [\cup]
\end{aligned}$$

2. $\llbracket t \rrbracket_q \subseteq \llbracket u \rrbracket_q$: Assume $x \in \llbracket t \rrbracket_q = \bigcup_{t \xrightarrow[\mathsf{trs(q)}]{*} t'} \mathscr{O}\langle t' \rangle$. Then $x \in \mathscr{O}t'$ for some t' with $t \xrightarrow[\mathsf{trs(q)}]{*} t'$. By confluence (Lemma 1),

$$\begin{tikzcd}
t \arrow[r, "*", "\mathsf{trs(q)}"'] \arrow[d, "\mathsf{trs(q)}"'] & t' \arrow[d, dashed, "\mathsf{trs(q)}"', "*"] \\
u \arrow[r, dashed, "*", "\mathsf{trs(q)}"'] & s
\end{tikzcd} .$$

By Lemma 2, $x \in \mathscr{O}s \Rightarrow x \in \bigcup_{u \xrightarrow[\mathsf{trs(q)}]{*} s} \mathscr{O}\langle s \rangle = \llbracket u \rrbracket_q$. □

Remark 2 If Definition 6 was changed so that the semantics of '*main* 0' w.r.t. the first program in Example 5 was the set

$$\{\,\textsc{cons}\ 0\ \bot\,,\ \textsc{cons}\ 0\ (\textsc{cons}\ (\textsc{succ}\ 0)\ \bot)\,,\ \ldots\}\ ,$$

then Lemma 3 would not hold. □

3.2.3 Leftmost rewrites

Since the semantics of a term is dependent on the observability of constructors from left to right, it is only necessary to use outermost-leftmost rewrites to get the meaning of a program. The following describes how I locate outermost-leftmost redexes.

Definition 8 Let $\bullet$ be a special variable not used in any term or program. Then I define *actives* r and *passives* e by the following grammar.

$$\begin{array}{lcl}
r & ::= & f\ t^n \\
 & | & g\ (c\ u^m)\ t^n \\
e & ::= & c\ o^m\ e\ t^n \\
 & | & b \\
o & ::= & c\ o^m \\
b & ::= & \bullet \\
 & | & g\ b\ t^n
\end{array}$$

I write $e[r]$ for $e[\bullet := r]$. □

Lemma 4 *Any ground term* t *is either a value or can by uniquely decomposed into an* e *and an* r *such that* $t = e[r]$.

PROOF. Let t be a non-value, and proceed by induction on the structure of t. If t has an outermost constructor, t must be of the form '$c\ o^m\ t_0\ t_1\ \ldots\ t_n$' where t_0 is a non-value. By the induction hypothesis, $t_0 = e[r]$ for some unique pair $\langle e, r\rangle$. Thus '$c\ o^m\ e\ t_1\ \ldots\ t_n$' is a unique passive.

If t does have an outermost constructor, t is either of the form '$f\ t^n$' or '$g\ t_0\ t^m$'. In the former case, t can only be decomposed into the unique passive $e = \bullet$ and active $r = t$, since passives cannot contain f-symbols. In the latter case, either t_0 has an outermost constructor or it has not. In the former case, the passive $e = \bullet$ and active $r = t$ is a unique decomposition since b's cannot contain '$g\ (c\ \ldots)\ \ldots$' forms. In the latter case, use the induction hypothesis. □

With the preceding definition I can locate outermost-leftmost redexes, and therefore I can easily describe an outermost-leftmost rewriting strategy.

Definition 9 For a program q, the *prudent outermost-leftmost rewriting* relation $\xrightarrow[\mathsf{fun(q)}]{} \subseteq \mathit{Term} \times \mathit{Term}$ is defined by

$$\mathsf{t} \xrightarrow[\mathsf{fun(q)}]{} \mathsf{u} \quad \Leftrightarrow \quad \mathsf{t} = \mathsf{e}[\mathsf{r}] \wedge \mathsf{r} \xrightarrow[\mathsf{redex(q)}]{} \mathsf{s} \wedge \mathsf{e}[\mathsf{s}] = \mathsf{u} \ .$$

□

The preceding outermost-leftmost rewriting strategy is not the standard way to define outermost-leftmost rewriting in general term-rewriting systems. The difference is that in *prudent* outermost-leftmost rewriting, stuck applications will stop rewriting of redexes to the right of the stuck application. I call it *prudent* because it only rewrites redexes that can contribute to the observable behaviour, which is witnessed be the following proposition.

Proposition 1 $[\![\mathsf{t}]\!]_{\mathsf{q}} = \bigcup_{\mathsf{t} \xrightarrow[\mathsf{fun(q)}]{*} \mathsf{t}'} \mathscr{O}\mathsf{t}'$.

The proof of the above proposition is based on the notion of *standard rewrite sequences.*

Definition 10 (Standard rewrite) Let q be a term-rewrite system and

$$\mathsf{R} = (\mathsf{t}_0 \xrightarrow[\mathsf{trs(q)}]{} \mathsf{t}_1 \xrightarrow[\mathsf{trs(q)}]{} \mathsf{t}_2 \xrightarrow[\mathsf{trs(q)}]{} \cdots)$$

be a rewrite sequence. Mark in every step of R all redex head symbols to the left of the head symbol of the contracted redex, and let marks be persistent in subsequent rewrites.

R is a *standard rewrite sequence* if in no step a redex is contracted with a marked head symbol. □

Lemma 5 (Klop's Standardisation Theorem) *If* q *is a left-normal orthogonal term-rewrite system and* $\mathsf{t} \xrightarrow[\mathsf{q}]{*} \mathsf{u}$*, then there is a standard rewrite sequence in* q *from* t *to* u*.*

PROOF. See Theorem 3.2.15 in Klop [55]. □

I am now in a position to reformulate every rewriting sequence as a series of prudent outermost-leftmost rewrites, followed by a series of non-prudent rewrites.

Definition 11 $\xrightarrow[\neg\mathsf{fun}(q)]{} = \xrightarrow[\mathsf{trs}(q)]{} \setminus \xrightarrow[\mathsf{fun}(q)]{}$. □

Lemma 6 *If* $t \xrightarrow[\mathsf{trs}(q)]{*} t'$, *then* $t \xrightarrow[\mathsf{fun}(q)]{*} t'' \xrightarrow[\neg\mathsf{fun}(q)]{*} t'$ (*for some* t'').

PROOF (sketch). Assume $t \xrightarrow[\mathsf{trs}(q)]{*} t'$. By Lemma 5 there is a standard rewrite sequence $t \xrightarrow[\mathsf{trs}(q)]{} t_1 \xrightarrow[\mathsf{trs}(q)]{} \cdots \xrightarrow[\mathsf{trs}(q)]{} t'$. Since $\xrightarrow[\mathsf{fun}(q)]{} \subseteq \xrightarrow[\mathsf{trs}(q)]{}$, there must be some t_n such that $t \xrightarrow[\mathsf{fun}(q)]{*} t_n$ and either $t_n = t'$ or $t_n \xrightarrow[\neg\mathsf{fun}(q)]{} t_{n+1}$. In the former case, the proof is concluded. In the latter case, $t_n = e[r]$ since t_n is not a value, but the contracted redex is not r. By Definition 8, there are no redexes to the left of r, and therefore each of the following contractions are to the right of r, and thus $t_n \xrightarrow[\neg\mathsf{fun}(q)]{*} t'$. □

The point of separating prudent from non-prudent rewrites is that non-prudent rewrites have no observable effect.

Lemma 7 $t \xrightarrow[\neg\mathsf{fun}(q)]{} t' \Rightarrow \mathscr{O}t = \mathscr{O}t'$.

PROOF (sketch). Assume $t \xrightarrow[\neg\mathsf{fun}(q)]{} t'$. Then $t = e[r]$ since t is not a value, but the contracted redex is not r. By Definition 8, there are no redexes to the left of r, and therefore the contraction is to the right of r. Since r by definition does not have an outermost constructor, it then follows by easy structural induction using Definition 6 that $\mathscr{O}t = \mathscr{O}t'$. □

PROOF of Proposition 1. By Definition 7, $[\![t]\!]_q = \bigcup_{t \xrightarrow[\mathsf{trs}(q)]{*} t'} \mathscr{O}t'$, so I need to prove that $\bigcup_{t \xrightarrow[\mathsf{trs}(q)]{*} t'} \mathscr{O}t' = \bigcup_{t \xrightarrow[\mathsf{fun}(q)]{*} t'} \mathscr{O}t'$. I will first prove inclusion from right to left ($\supseteq$).

1. $u \in \bigcup_{t \xrightarrow[\mathsf{fun}(q)]{*} t'} \mathscr{O}t'$ [Assumption]
2. $\exists t' \; (t \xrightarrow[\mathsf{fun}(q)]{*} t')$ [1]
3. $u \in \mathscr{O}t'$ [*do.*]
4. $t \xrightarrow[\mathsf{trs}(q)]{*} t'$ [2 & $\xrightarrow[\mathsf{fun}(q)]{} \subseteq \xrightarrow[\mathsf{trs}(q)]{}$]
5. $u \in \bigcup_{t \xrightarrow[\mathsf{trs}(q)]{*} t'} \mathscr{O}t'$ [4 & 3]

The inclusion from left to right ($\subseteq$) goes as follows.

1. $u \in \bigcup_{t \xrightarrow[\mathsf{trs}(q)]{*} t'} \mathscr{O}t'$ [Assumption]
2. $\exists t' \; (\, t \xrightarrow[\mathsf{trs}(q)]{*} t' \,)$ [1]
3. $u \in \mathscr{O}t'$ [*do.*]
4. $\exists t'' \; (\, t \xrightarrow[\mathsf{fun}(q)]{*} t'' \xrightarrow[\neg\mathsf{fun}(q)]{*} t' \,)$ [2 & Lemma 6]
5. $u \in \mathscr{O}t''$ [4 & 3 & Lemma 7]
6. $u \in \bigcup_{t \xrightarrow[\mathsf{fun}(q)]{*} t''} \mathscr{O}t''$ [5 & 4]

□

In fact, the combination of observability and prudent leftmost-outermost rewriting is not just appropriate for non-terminating programs. If a term t can be rewritten (w.r.t. program q) to a value v, then $v \in [\![t]\!]_q$ (indeed $v = \max[\![t]\!]_q$, were the size of $\perp$ defined to be nought). Conversely, if $v \in [\![t]\!]_q$, then $t \xrightarrow[\mathsf{fun}(q)]{*} v$. In this respect, the notion of observability subsumes the usual notion of evaluation to normal form.

With respect to the quantitative aspects of the programming language, I will define the running time of a program–term pair $\langle q \, , \, t_0 \rangle$ as the length of the rewrite sequence

$$t_0 \xrightarrow[\mathsf{fun}(q)]{} t_1 \xrightarrow[\mathsf{fun}(q)]{} \cdots \; .$$

This measure is rather crude, but it is probably the best that can defined when working with languages at the present level of abstraction. The other big influence on the *actual* running time of programs of this kind, is space consumption. I will, however, not deal with the space consumption of computations, let alone garbage collection issues, since that would require a much more detailed view of the program execution process. There is a substantial body of literature that deals with the compilation process, garbage collection and profiling of non-strict languages, see for instance Peyton Jones [51], Johnsson [45], Plasmeijer and van Eekelen [86], Sestoft [99], or Boquist [11].

3.3 Primitive values and operators

For practical purposes, it is usually necessary to have a set of primitive values (like integers, floats, etc.) and a set of predefined operators on these values (like

'+', '==', etc.). The idea is that such functions can be implemented directly into hardware, and I will therefore require that predefined operations can be computed in constant time.

I will occasionally extend the language to include such primitives, and extend the rewriting rules as follows. A term is a redex if it has a root labelled by a predefined operator $\oplus$ and observables $\langle o^n \rangle$ as children. Such a redex rewrites to whatever the predefined operator prescribes (denoted by '$[\![\oplus]\!]\ o^n$') by a so-called δ-rule,[2] denoted $t \xrightarrow[\delta]{} t'$ to indicate that the reduction is independent of the program at hand. I will use infix notation (i.e., '$t \oplus u$' instead of prefix notation '$\oplus\ t\ u$') when it is convenient.

Example 6 Assume program $q = \{\mathit{inc}\ x = x + 1\}$ and operator '+' that behaves as expected on integers. Then

$$(\mathit{inc}\ 2){+}(\mathit{inc}\ 2) \xrightarrow[\mathsf{trs}(q)]{} (2{+}1){+}(\mathit{inc}\ 2) \xrightarrow[\delta]{} 3{+}(\mathit{inc}\ 2) \xrightarrow[\mathsf{trs}(q)]{} 3{+}(2{+}1) \xrightarrow[\delta]{} 3{+}3 \xrightarrow[\delta]{} 6$$

is a rewrite sequence for '$(\mathit{inc}\ 2) + (\mathit{inc}\ 2)$'. □

When I allow primitives in the language, Definition 8 has to be extended by

$$\begin{array}{lll} r & ::= & \oplus\ o^n \mid \ldots \\ e & ::= & \oplus\ o^n\ e\ t^n \mid \ldots \end{array}$$

to accommodate for predefined operators. The definition of $\xrightarrow[\mathsf{fun}(q)]{}$ has to adjusted accordingly.

3.4 Expressiveness

Although the little language defined in this chapter can express all computable functions, it somewhat limits the expressiveness of the programmer in practice. Put differently,

> "To say that any reasonable function can be expressed by some program is not to say that it can be expressed by the most reasonable program. [...] Moreover, it is likely that certain important functions cannot be expressed by their most efficient algorithms."
>
> *J. C. Reynolds [87]*

[2] The δ originates from λ-calculus where there are also α-, β, and η-rules.

This is indeed true for the language presented here. However, the language could, in principle, be seen as a low-level representation used in a compiler for a higher-order language by following the recipe

1. Compile complex pattern matching into flat pattern matching (see Augustsson [4]).
2. Remove local scope by λ-lifting locally defined functions (see Hughes [44] and Johnsson [45]).
3. Use a flow analysis to decide which functions are called by higher-order functions (see e.g., Faxén [28]).
4. Either in-line functions or represent them as tags, based on the above analysis, thus removing the need for higher-order functions.

Alternatively, for some programs, the last two steps can be replaced by the *higher-order removal* as presented by Chin and Darlington [18].

Chapter 4

Deforestation

In this chapter we shall give an account of traditional *deforestation*, an algorithm which eliminates intermediate data structures from functional programs. A number of different formulations of essentially the same algorithm exists in the literature — the difference being mainly the formulation of folding steps.

Many researchers have studied the problem of intermediate data structures since early 1980s and developed techniques for elimination of them. Deforestation is one of these techniques, introduced by Philip Wadler in 1990, inspired by earlier work of the same author.

4.1 Operational semantics

To ease the presentation of deforestation, I will here present an operational semantics for our programming language based on execution traces.

Definition 12 Given a program $\mathsf{q} \in \mathit{Program}$, the relations $\xrightarrow[\mathsf{outer(q)}]{}$ and $\xrightarrow[\mathsf{inner(q)}]{}$ $\subseteq \mathit{Term} \times \mathit{Term}$ are defined as the smallest relations satisfying the inference system in Figure 4.1.

□

The relation $\xrightarrow[\mathsf{outer(q)}]{}$ is an operational formulation of $\xrightarrow[\mathsf{fun(q)}]{}$ (cf. Chapter 3), but differs from $\xrightarrow[\mathsf{fun(q)}]{}$ in that $\xrightarrow[\mathsf{outer(q)}]{}$ simply throws away the outermost constructors and evaluates each of the constructor's arguments independently. Such

$$n \geq 0, m \geq 0, t \in \mathit{Term}, x \in \mathit{Variable}, v \in \mathit{Value},$$
$$c \in \mathit{Constructor}, f \in \mathit{Function}, g \in \mathit{Matcher}$$

$$(\textsc{fcall})\ \frac{f\ x^n \overset{q}{=} t}{f\ t^n \xrightarrow[\text{inner}(q)]{} t[\,(x_{.} := t_{.})^n\,]}$$

$$(\textsc{match})\ \frac{g\,(c\ x^m)\ x_{m+1}\ldots x_n \overset{q}{=} t}{g\,(c\ t^m)\ t_{m+1}\ldots t_n \xrightarrow[\text{inner}(q)]{} t[\,(x_{.} := t_{.})^n\,]}$$

$$(\textsc{gdive})\ \frac{t \xrightarrow[\text{inner}(q)]{} t' \qquad g\ p\ x^n \overset{q}{=} u}{g\ t\ t^n \xrightarrow[\text{inner}(q)]{} g\ t'\ t^n}$$

$$(\textsc{indep})\ \frac{}{c\ t^n \xrightarrow[\text{outer}(q)]{} t_i}\ i \in \{1 \ldots n\} \qquad (\textsc{outin})\ \frac{t \xrightarrow[\text{inner}(q)]{} t'}{t \xrightarrow[\text{outer}(q)]{} t'}$$

Figure 4.1: Small step, non-strict, forgetful semantics.

a forgetful relation is easier to work with when formulating program transformations. I will now work out the exact relation between the $\xrightarrow[\text{outer}(q)]{}$- and $\xrightarrow[\text{fun}(q)]{}$-semantics to show that nothing is lost nor gained by working with $\xrightarrow[\text{outer}(q)]{}$ instead of $\xrightarrow[\text{fun}(q)]{}$.

Remark 3 The operational semantics defined by $\xrightarrow[\text{outer}(q)]{}$ ignores primitive operators. In the rest of this chapter, I will ignore operators since their treatment is straightforward but clutters the presentation of deforestation.

Definition 13 (Trace) Given $\to \subseteq \mathit{Term} \times \mathit{Term}$ and $t \in \mathit{Term}$, the *trace of* t *with respect to* $\to$ is the possibly infinite tree $\Delta_t^{\to}$ defined by

$$\Delta_t^{\to} = t\ (\Delta_{t_{.}}^{\to})^n$$

where $\{\,t^n\,\} = \{\,t' \mid t \to t'\,\}$. □

Definition 14 (Computation trace) Given some program $q \in \mathit{Program}$ and ground term $t \in \mathit{Term}$.

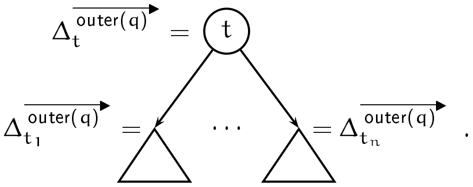
1. The *computation trace of* t *with respect to* q is defined to be $\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$. In case $t = c\ u^n$, I will assume that $t_i = u_i$, that is, the ordering of children in the trace tree follows the ordering of constructor arguments in t.

2. Given a *finite* computation trace $\Delta_t^{\overrightarrow{\mathsf{outer}(q)}} = t\ \left(\Delta_{t_\cdot}^{\overrightarrow{\mathsf{outer}(q)}}\right)^n$, the *value of* $\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$ is denoted $V\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$ and defined by

$$V\Delta_t^{\overrightarrow{\mathsf{outer}(q)}} \overset{\text{def}}{=} \begin{cases} c\ (V\Delta_{t_\cdot}^{\overrightarrow{\mathsf{outer}(q)}})^n & \text{if } t = c\ t^n \\ V\Delta_{t_1}^{\overrightarrow{\mathsf{outer}(q)}} & \text{otherwise.} \end{cases}$$

   In the otherwise-clause, necessarily $n = 1$ since evaluation by rule OUTIN is deterministic. □

Graphically, the computation trace will look like this:


$\Delta_t^{\overrightarrow{\mathsf{outer}(q)}} =$
t
$\Delta_{t_1}^{\overrightarrow{\mathsf{outer}(q)}} =$
...
$= \Delta_{t_n}^{\overrightarrow{\mathsf{outer}(q)}}$
.

Example 7 Consider again the append program

$$\begin{array}{ll} \mathit{append}\ []\ \mathit{ys} & = \mathit{ys} \\ \mathit{append}\ (x : \mathit{xs})\ \mathit{ys} & = x : (\mathit{append}\ \mathit{xs}\ \mathit{ys})\ . \end{array}$$

Using the abbreviation $[\,t_1, \ldots, t_n\,]$ for a list $t_1 : (\cdots : (t_n : []))$, the computation trace for '$\mathit{append}\ [1,2]\ [3,4]$' is shown in Figure 4.2. The value of this tree is $1 : (2 : (3 : (4 : [])))$. □

Intuitively, $V\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$ collects all the constructors from a computation trace to construct the final value of the computation. This intuition can be formalised.

Lemma 8

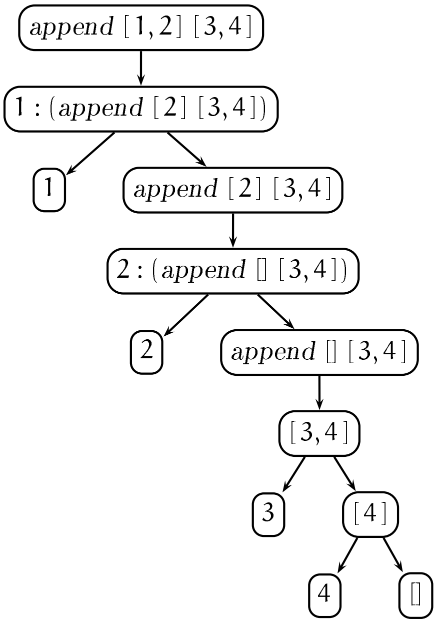


Figure 4.2: Computation trace

1. $\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$ *is finite* $\Leftrightarrow \exists n \in \mathbb{N}, v \in \mathit{Value}\left(t \xrightarrow[\mathsf{fun}(q)]{n} v \right)$.

2. $\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$ *is finite* $\Rightarrow \max([\![t]\!]_q) = V\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$.

In other words, the computation trace $\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$ is finite if, and only if, the sequence $t \xrightarrow[\mathsf{fun}(q)]{} \ldots$ is; and, in this case the final result v of the evaluation of t is exactly the value $V\Delta_t^{\overrightarrow{\mathsf{outer}(q)}}$ of the computation trace.

The following style of presentation will be used throughout this text to present several program transformations techniques.

4.2 Transformation Trace

There are several differences between *evaluation* and *transformation*. One difference is that the former concerns *ground terms* (i.e., terms without variables), whereas the latter concerns terms in general, in particular terms containing free variables. For that purpose, define the following subclass of terms.

Definition 15 (Passive term) A *passive term* is defined by the grammar

$$\mathit{Passive} \ni a ::= x \mid c\ a^n \ ,$$

where $x \in \mathit{Variable}$ and $c \in \mathit{Constructor}$. □

In fact, the core of the deforestation algorithm (to be introduced in the following), can be seen as a natural generalisation of a computation trace by modifying the inference system in Figure 4.1 to take terms with variables into account.

Definition 16 (Deforestation unfold) Given $q \in \mathit{Program}$, the relation $\xrightarrow[\mathsf{deforest}(q)]{}$ $\subseteq \mathit{Term} \times \mathit{Term}$ is defined as $\xrightarrow[\mathsf{outer}(q)]{}$ by adding the following rule to the inference system in Figure 4.1:

$$(\textsc{spec})\ \frac{g\ (c\ y^m)\ x^n \overset{q}{=} t \qquad (y^m \text{ fresh})}{g\ y\ t^n \xrightarrow[\mathsf{inner}(q)]{} t[\,(x. := t.)^n\,]}$$

□

Rule SPEC deals with "stuck" pattern matchers: When the first argument to a pattern-matching function g is a variable y, the variable y is assumed to be any of the different constructors seen in the left-hand side of the definition of g. The relation $\xrightarrow[\mathsf{deforest(q)}]{}$ is thus non-deterministic.

I now generalise the notion of computation trace, which was defined on top of $\xrightarrow[\mathsf{outer(q)}]{}$, to the notion of *transformation trace* which will be defined on top of $\xrightarrow[\mathsf{deforest(q)}]{}$ analogously.

Definition 17 (transformation trace) Given $\mathsf{q} \in \mathit{Program}$ and $\mathsf{t} \in \mathit{Term}$, the *transformation trace of* t is the tree $\Delta_{\mathsf{t}}^{\xrightarrow{\mathsf{deforest(q)}}}$ with the following requirements.

1. For a term $\mathsf{c}\ \mathsf{t}^{\mathsf{n}}$, the children must appear in the order corresponding to the constructor arguments.

2. For a term $\mathsf{g}\ \mathsf{x}\ \mathsf{t}^{\mathsf{n}}$, the children must appear in the order corresponding to the patterns defining g. □

Example 8 Consider again the append program. The transformation trace for '$\mathit{append}\ xs\ ys$' is the infinite tree depicted in Figure 4.3. I have added labels on some of the edges to indicate which definition of 'append' that was used: When the term '$\mathit{append}\ ys\ zs$', say, is matched against the first alternative ($\mathit{append}\ []\ ys$), I label the edge with '$ys = []$'; and when matched against the second alternative ($\mathit{append}\ (x : xs)\ ys$), I label the edge with '$ys = y' : ys'$'. □

Example 9 The transformation trace for '$\mathit{append}\ (\mathit{append}\ xs\ ys)\ zs$' is the infinite tree depicted in Figure 4.4. □

A transformation trace for a term t is, in a certain sense, a model of all computations with t (but I will not bother to make this precise at this stage). From a program-transformation perspective, the point is that the transformation trace displays certain regularities, which can be used to extract a new and more efficient way of computing, for instance, '$\mathit{append}\ (\mathit{append}\ xs\ ys)\ zs$', as you will see in the following.

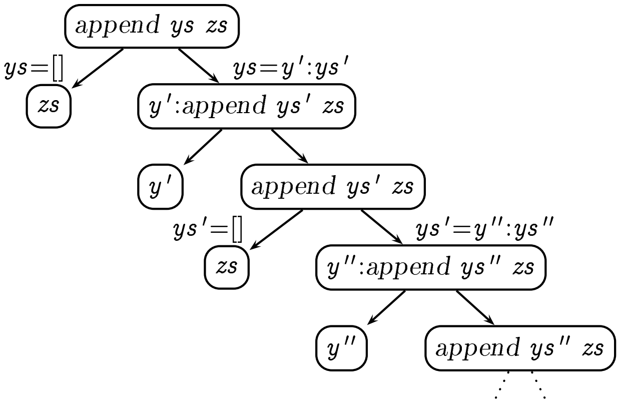


Figure 4.3: Transformation trace of *append ys zs*.

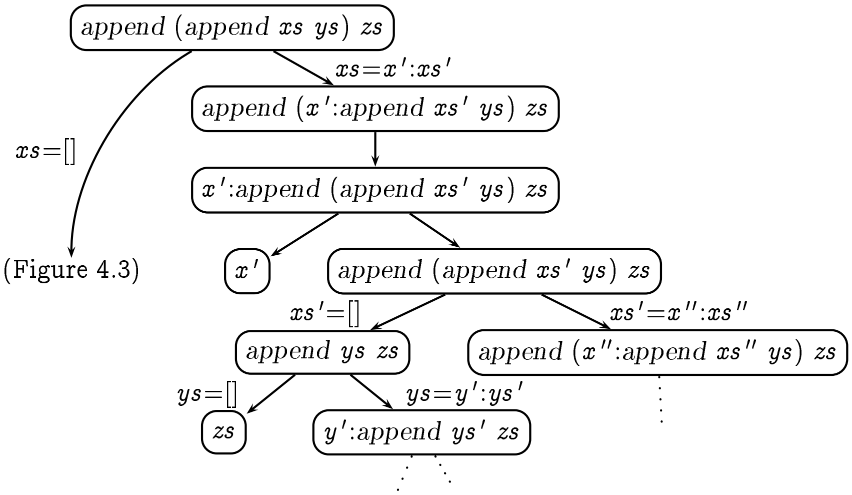


Figure 4.4: Transformation trace of *append* (*append xs ys*) *zs*.

4.2.1 From Infinite Trees to Finite Trees

From the previous examples, you have seen that unfolding a program may produce infinite transformation traces. I now give a principle by which a class of infinite transformation traces may be mapped to corresponding finite trees where certain branches are cut off. The principle is: For every node t, if there exists an ancestor t' such that t is identical to t' up to choice of variable names, I can remove all children of node t.

Definition 18 (Renaming) Let $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ be a bijection from the variable set $\{x^n\}$ to the variable set $\{y^n\}$. Then the substitution $\theta = [x_1{:=}y_1, \ldots, x_n{:=}y_n]$ is a *renaming*; moreover, if $t = t'\theta$, then I will say that t is a *renaming* of t', denoted $t \equiv t'$. □

Definition 19 (Tree pruning) Given a transformation trace $\tau = \Delta_t^{\overrightarrow{\mathsf{deforest}(q)}}$, for any node $\eta \in \mathscr{D}\tau$ where there exists an ancestor $\mu <_\tau \eta$ such that $\tau\eta \equiv \tau\mu$, I can replace τ with $\tau[\eta{:=}(t\eta)]$. □

The rationale behind the above *pruning* is that the computations represented by η are the same as those represented by μ.

Example 10 Exhaustive pruning of the transformation trace tree for the double-append program (cf. Figure 4.4) is the finite tree depicted in Figure 4.5.

□

4.2.2 From Finite Trees to New Programs

From a finite tree as in the previous example, I can recover a new program as described below. To separate program code from the algorithm consuming the tree and producing the code, I will use Quine's quasi-quotes $\ulcorner$ and $\urcorner$ to delimit code that comes from the original program or will end up in the derived program. For convenience, I will assume that a program is simply a set of program-code fragments, each representing a definition of a function., and by $\cup$ I will denote concatenation of code.

A program is extracted by traversing the transformation tree, approximately generating a new function for each node in the tree. To be able to generate

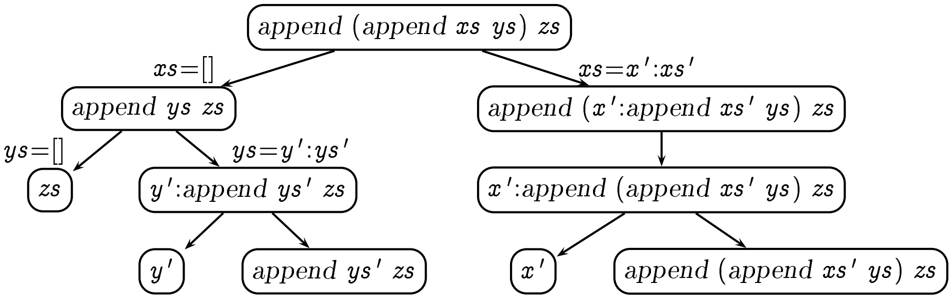


Figure 4.5: Pruned transformation trace of *append* (*append* *xs* *ys*) *zs*.

recursive calls in the new program, each node η in the transformation tree is associated with a unique function symbol indexed by η (i.e., f_η or g_η, depending on whether the function has patterns or not), the idea being that when a leaf which represents a call to an ancestor is met, a call to the right function can be generated.

To find out which kind of program construct I should generate, I divide the label of each node into two parts, the *active* part and the *passive* part. The active part constitutes the sub-term that gave rise to the unfolding, whereas the passive part is the surrounding term. There will then be exactly one way to decompose a label into the two parts.

Definition 20 (Decomposition) If $t \in \mathit{Term}$ is not of the form $c\ t^n$, it can be decomposed as $e[r]$ where e and r are defined by the grammars below and $e[r]$ is denotes replacing the hole $\bullet$ in e by r.

$$\begin{array}{lll lll lll} e & ::= & \bullet & r & ::= & f\ t^n & o & ::= & x \\ & \mid & g\ e\ t^n & & \mid & g\ o\ t^n & & \mid & c\ t^n\ . \end{array}$$

□

Definition 21 (Program extraction) Given a finite transformation trace τ, the program extracted from τ is

$$\{\ulcorner \mathit{main}\ x^n = t \urcorner\} \cup q \ \ \textbf{where}\ \ \langle t\,,\ q\rangle := \mathrm{cogen}\ \tau\ \langle\rangle$$
$$\langle x^n\rangle = \mathscr{V}\,t\ \ ,$$

where 'cogen' is defined by the algorithm in Figure 4.6. □

```
cogen $\tau\ \eta$ =
 let $\langle x^n \rangle = \mathscr{V}(\tau\eta)$
 in if $\eta \in \text{leaves}_\tau$
   then if $\tau\eta \in$ Passive
       then $\langle \ulcorner \tau\eta \urcorner, \varnothing \rangle$
       else let $\mu$ suchthat $\mu <_\tau \eta \wedge \tau\mu \equiv \tau\eta$
           in $\langle \ulcorner h_\mu\ x^n \urcorner, \varnothing \rangle$
   else let $\langle \langle \ulcorner t. \urcorner, q. \rangle^m \rangle$ = map (cogen $\tau$) (children$_\tau \eta$)
       in if $\tau\eta = c \ldots$ then $\left\langle \ulcorner c\ t^m \urcorner, \bigcup_{i \le m} q_i \right\rangle$
         else if $\tau\eta = \ulcorner e[g\ x \ldots] \urcorner$ (for some $e$)
         then $\left\langle \ulcorner g_\eta\ x^n \urcorner, \bigcup_{i \in \{0 \ldots m\}} q_i \right\rangle$
              where $q_0 = \{$
                  $\ulcorner g_\eta\ (c_1\ y^k)\ x_2\ \ldots\ x_n = t_1 \urcorner$
                      where $\langle y^k \rangle = \mathscr{V} t_1 \dot{-} \mathscr{V}(\tau\eta)$
                          $\vdots$
                  $\ulcorner g_\eta\ (c_m\ y^k)\ x_2\ \ldots\ x_n = t_m \urcorner$
                      where $\langle y^k \rangle = \mathscr{V} t_m \dot{-} \mathscr{V}(\tau\eta)$
                  where $c^m$ are the constructors from
                             the definition of $g$
              $\}$
         else $\langle \ulcorner f_\eta\ x^n \urcorner, \{\ulcorner f_\eta\ x^n = t_1 \urcorner\} \cup q_1 \rangle$
```

Figure 4.6: Extracting a program from a finite transformation trace.

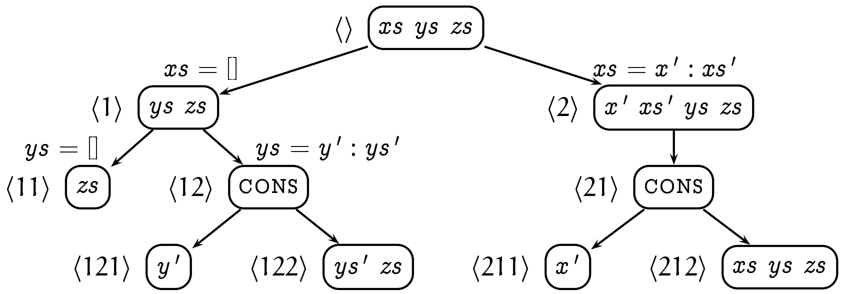


Figure 4.7: Operative parts of a transformation trace tree

Example 11 From the finite tree in Example 10, I can get the program as follows. Each internal node produces a new function definition, and the leaves produce a variable or a recursive call. The operative part of the tree is illustrated in Figure 4.7. The new program will then be

$$\begin{aligned}
&\mathit{main}\ xs\ ys\ zs &&= g_{\langle\rangle}\ xs\ ys\ zs\\
&g_{\langle\rangle}\ []\ ys\ zs &&= g_{\langle 1\rangle}\ ys\ zs\\
&g_{\langle\rangle}\ (x' : xs')\ ys\ zs &&= f_{\langle 2\rangle}\ x'\ xs'\ ys\ zs\\
&f_{\langle 2\rangle}\ x'\ xs'\ ys\ zs &&= x' : (g_{\langle\rangle}\ xs'\ ys\ zs)\\
&g_{\langle 1\rangle}\ []\ zs &&= zs\\
&g_{\langle 1\rangle}\ (y' : ys')\ zs &&= y' : (g_{\langle 1\rangle}\ ys'\ zs)
\end{aligned}$$

The intermediate function $f_{\langle 2\rangle}$ can be eliminated by a simple unfolding step in the new program. □

4.3 Deforestation

I have now introduced enough ideas and terminology to present an abstract version of the deforestation algorithm.

Definition 22 (Deforestation) Given $\mathrm{q} \in \mathit{Program}$ and $\mathrm{t} \in \mathit{Term}$.

1. Construct the transformation trace $\tau = \Delta_{\mathrm{t}}^{\overrightarrow{\mathsf{deforest}(\mathsf{q})}}$.

2. Prune τ by using Definition 19 to get a finite tree τ'.

3. Construct a new program from τ' by using Definition 21. □

The algorithm terminates provided the set $\{\, t' \mid t \xrightarrow[\mathsf{deforest(q)}]{*} t' \,\}$ contains only finitely many different terms modulo renaming. This is the same as saying that the generated transformation trace contains only finitely many different subtrees modulo variable renaming.

Definition 23 (Finite modulo renaming) A set of terms $S \subseteq \mathit{Term}$ is *finite modulo renaming* if the set of equivalence classes with respect to the renaming equivalence relation $(\equiv)$ is finite, that is, if $\{\, [s]_{\equiv} \mid s \in S \,\}$ is finite. □

Example 12 Consider the program

$$\begin{array}{lll}
\textbf{data}\ \text{Tree}\ a & = & \text{LEAF}\ a \mid \text{BRANCH}\ (\text{Tree}\ a)\ (\text{Tree}\ a) \\
\mathit{main}\ x & = & \mathit{flip}\ (\mathit{flip}\ x) \\
\mathit{flip}\ (\text{LEAF}\ n) & = & \text{LEAF}\ n \\
\mathit{flip}\ (\text{BRANCH}\ x\ y) & = & \text{BRANCH}\ (\mathit{flip}\ y)\ (\mathit{flip}\ x)\ \ .
\end{array}$$

Deforestation of this program will give the finite transformation trace shown in Figure 4.8, which yields the new program

$$\begin{array}{lll}
\mathit{main}\ z & = & g_{\langle\rangle}\ z \\
g_{\langle\rangle}\ (\text{LEAF}\ n) & = & f_{\langle 1\rangle}\ n \\
g_{\langle\rangle}\ (\text{BRANCH}\ x\ y) & = & f_{\langle 2\rangle}\ x\ y \\
f_{\langle 1\rangle}\ n & = & \text{LEAF}\ n \\
f_{\langle 2\rangle}\ x\ y & = & \text{BRANCH}\ (g_{\langle\rangle}\ x)\ (g_{\langle\rangle}\ y)\ \ .
\end{array}$$

This new program only traverses the tree once. The intermediate functions $f_{\langle 1\rangle}$ and $f_{\langle 2\rangle}$ can be eliminated by a simple unfolding step in the new program. □

In the rest of this chapter I shall be concerned with the properties of deforestation in several respects, mainly *termination*. It is a reasonable question to ask "does the deforestation algorithm always terminate?" The answer is *no*. It turns out that there exist programs q and terms t such that $\{\, t' \mid t \xrightarrow[\mathsf{deforest(q)}]{*} t' \,\}$ contains infinitely many different terms, even if I identify terms that differ only in the choice of names for variables. Another way of putting this is to say that

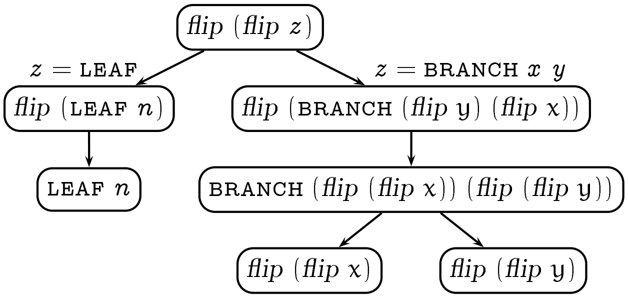


Figure 4.8: Transformation trace of double flip.

the transformation trace $\Delta_t^{\overrightarrow{\mathsf{deforest}(q)}}$ contains infinitely many different terms (even if I identify renamings). Yet another formulation is that $\Delta_t^{\overrightarrow{\mathsf{deforest}(q)}}$ is *irregular*, that is, contains infinitely many different subtrees (even if ...). All these equivalent formulations amount to saying: Step 2 of the algorithm cannot succeed.

4.4 The Forms of Non-termination

I now proceed to examine some examples of programs which lead to non-termination.

4.4.1 Accumulating parameter

The following example illustrates a phenomenon called *the accumulating parameter*. Consider the program

$$\begin{aligned}
&\mathit{rev}\ \mathit{xs} && = \mathit{rr}\ \mathit{xs}\ [\,] \\
&\mathit{rr}\ [\,]\ \mathit{ys} && = \mathit{ys} \\
&\mathit{rr}\ (x : \mathit{xs})\ \mathit{ys} && = \mathit{rr}\ \mathit{xs}\ (x : \mathit{ys})\ .
\end{aligned}$$

The function *rev* returns its argument list *reversed* by calling an auxiliary function *rr* using an accumulating parameter. In Figure 4.9 you can see what happens when I try to transform the term '*rev l*'.

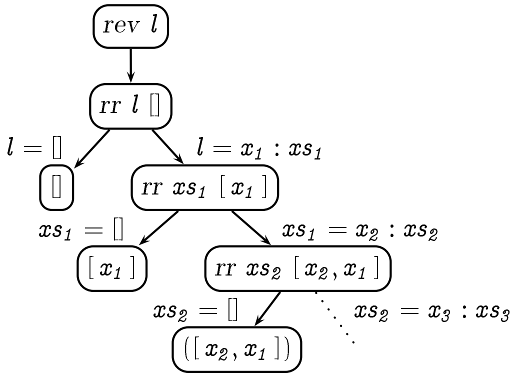


Figure 4.9: Accumulating parameter problem. Repeated unfolding of calls to *rev* piles up elements in the second parameter of *rev*.

The construction of the tree never ends, because I never encounter a term which is an instance of a previous term. The problem is that the calls

$$rr\ xs\ ys$$

are instantiated by $[\,xs := x' : xs'\,]$ and unfolded to

$$rr\ xs'\ (x' : ys)\ ,$$

so the second argument is growing unboundedly due to its accumulating parameter.

4.4.2 Obstructing function

Now consider the following program:

$$\begin{aligned} &rev\ [] &&= [] \\ &rev\ (x : xs) &&= append\ (rev\ xs)\ [\,x\,]\ . \end{aligned}$$

The function *rev* again reverses its argument list, now by using *append* (defined as usual). The transformation tree seen in Figure 4.10 of term '*rev l*' displays a phenomenon called *the obstructing function call.*

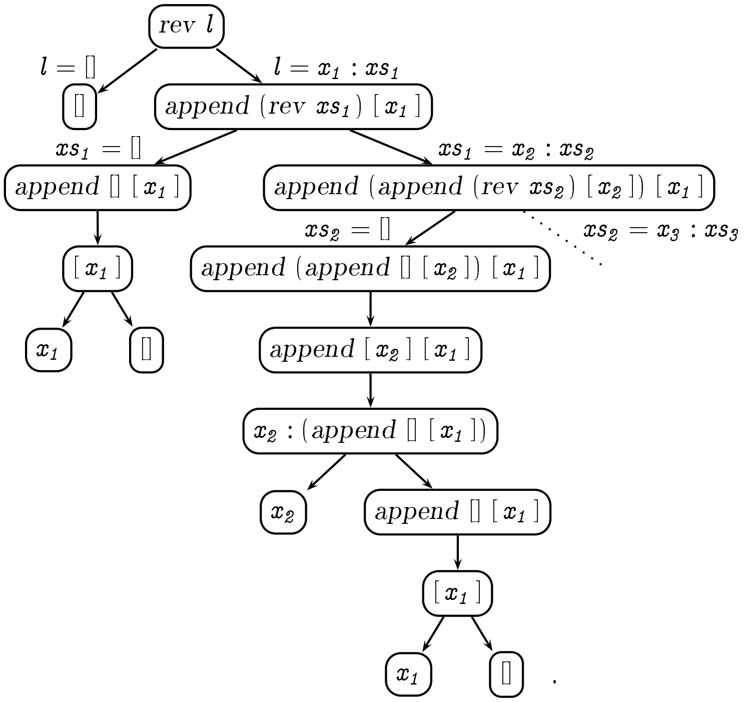


Figure 4.10: Obstructing function problem. Repeated unfolding of calls to *rev* piles up calls to *append*.

Again, construction of a finite tree fails to end because I never encounter a term which is an instance of a previous term. The problem is that the calls

$$\mathit{rev}\ \mathit{xs}$$

are instantiated by $[\,\mathit{xs} := x' : \mathit{xs}'\,]$ and unfolded to

$$\mathit{append}\ (\mathit{rev}\ \mathit{xs}')\ [\mathrm{x}'\,]$$

which will never be transformed to a term with an outermost constructor that the surrounding *append* call can consume, so the nesting of calls grows unboundedly due to the recursive call to *rev* in *rev*.

Remark 4 The same problem surfaces for primitive operators: An inner operator can block the evaluation of an outer operator.

4.5 Treeless Terms and Programs

It turns out that the above two problems are the *only* problems that can occur, which is known as *Wadler's Treeless Theorem.* This theorem gives a *syntactic* characterisation of a class of programs for which deforestation is guaranteed to terminate, namely the *treeless programs.*

Definition 24 (Treeless) The set of *Treeless* programs, denoted *Program*$^-$, is defined by the following grammar.

$$\begin{array}{rcrcl}
\mathit{Program}^- & \ni & \mathrm{q} & ::= & \mathrm{d}^{\mathrm{n}} \\
 & & \mathrm{d} & ::= & \mathrm{f}\ \mathrm{x}^{\mathrm{n}} \overset{\mathrm{q}}{=} \mathrm{t} \\
 & & & \mid & (\mathrm{g}\ \mathrm{p}.\ \mathrm{x}^{\mathrm{n}} = \mathrm{t}.)^{\mathrm{m}} \\
 & & \mathrm{p} & ::= & \mathrm{c}\ \mathrm{x}^{\mathrm{n}} \\
\mathit{Term}^- & \ni & \mathrm{t} & ::= & \mathrm{x} \\
 & & & \mid & \mathrm{c}\ \mathrm{t}^{\mathrm{n}} \\
 & & & \mid & \mathrm{f}\ \mathrm{x}^{\mathrm{n}} \\
 & & & \mid & \mathrm{g}\ \mathrm{x}_0\ \mathrm{x}^{\mathrm{n}}\ \ .
\end{array}$$

□

In words, a term is treeless if all function arguments are variables (and it does not contain primitive operators).

Theorem 1 (Wadler's Treeless Theorem) *For all* $\mathsf{q} \in \mathit{Program}^{-}$ *and* $\mathsf{t} \in$ *Term, the set* $\{\, \mathsf{t}' \mid \mathsf{t} \xrightarrow[\mathsf{deforest(q)}]{*} \mathsf{t}' \,\}$ *is finite modulo renaming.*

First I will show that the *nesting of function calls* decreases by deforestation on treeless programs. Then I will show that deforestation always terminates when I only consider special subclasses of terms and treeless programs. I will then show how general terms and treeless programs can be transformed into these special subclasses, and that deforestation on these transformed terms and programs will terminate if and only if deforestation terminates on the original terms and treeless programs.

Definition 25 (Nesting of calls) Let $\mathscr{N} \in \mathit{Term} \to \mathbb{N}$ be defined by

$$\begin{aligned}
\mathscr{N}(\mathsf{x}) &\overset{\text{def}}{=} 0 \\
\mathscr{N}(\mathsf{c}\ \mathsf{t}^{\mathsf{n}}) &\overset{\text{def}}{=} \max\{\, \mathscr{N}(\mathsf{t}_1) \ldots \mathscr{N}(\mathsf{t}_{\mathsf{n}}) \,\} \\
\mathscr{N}(\mathsf{f}\ \mathsf{t}^{\mathsf{n}}) &\overset{\text{def}}{=} 1 + \max\{\, \mathscr{N}(\mathsf{t}_1) \ldots \mathscr{N}(\mathsf{t}_{\mathsf{n}}) \,\} \\
\mathscr{N}(\mathsf{g}\ \mathsf{t}^{\mathsf{n}}) &\overset{\text{def}}{=} 1 + \max\{\, \mathscr{N}(\mathsf{t}_1) \ldots \mathscr{N}(\mathsf{t}_{\mathsf{n}}) \,\}
\end{aligned}$$

where $\max \varnothing \overset{\text{def}}{=} 0$ by convention. □

Lemma 9 $\forall \{\, \mathsf{t}\,,\ \mathsf{t}^{\mathsf{n}} \,\} \subseteq \mathit{Term}\ \big(\mathscr{N}(\mathsf{t}) + \max\big\{\, (\mathscr{N}(\mathsf{t}.))^{\mathsf{n}} \,\big\} \geq \mathscr{N}(\mathsf{t}[\,(\mathsf{x}. := \mathsf{t}.)^{\mathsf{n}}\,])\big)$.

PROOF. By induction on the structure of t.

1. $\mathsf{t} = \mathsf{x}$: Either $\mathsf{x} \notin \{\, \mathsf{x}^{\mathsf{n}} \,\}$ or $\mathsf{x} = \mathsf{x}_{\mathsf{i}}$ for some $\mathsf{i} \in \{\, 1 \ldots \mathsf{n} \,\}$.

   (a) If $\mathsf{x} \notin \{\mathsf{x}^{\mathsf{n}}\}$ then $\mathscr{N}(\mathsf{t}) + \max\big\{\, (\mathscr{N}(\mathsf{t}.))^{\mathsf{n}} \,\big\} = \max\big\{\, (\mathscr{N}(\mathsf{t}.))^{\mathsf{n}} \,\big\} \geq 0 = \mathscr{N}(\mathsf{t}) = \mathscr{N}(\mathsf{t}[\,(\mathsf{x}. := \mathsf{t}.)^{\mathsf{n}}\,])$.

   (b) If $\mathsf{x} = \mathsf{x}_{\mathsf{i}}$ then $\mathscr{N}(\mathsf{t}) + \max\big\{\, (\mathscr{N}(\mathsf{t}.))^{\mathsf{n}} \,\big\} = \max\big\{\, (\mathscr{N}(\mathsf{t}.))^{\mathsf{n}} \,\big\} \geq \mathscr{N}(\mathsf{t}_{\mathsf{i}}) = \mathscr{N}(\mathsf{t}[\,(\mathsf{x}. := \mathsf{t}.)^{\mathsf{n}}\,])$.

2. $t = c\ u^m$: Then

$$\begin{aligned}
\mathcal{N}(t) + \max\{\,(\mathcal{N}(t.))^n\,\} &= \max\{\,(\mathcal{N}(u.))^m\,\} + \max\{\,(\mathcal{N}(t.))^n\,\} \\
&= \max\left\{\begin{array}{c} \mathcal{N}(u_1) + \max\{\,(\mathcal{N}(t.))^n\,\} \\ \vdots \\ \mathcal{N}(u_m) + \max\{\,(\mathcal{N}(t.))^n\,\} \end{array}\right\} \\
\text{(by induction hypothesis)} \quad &\geq \max\left\{\begin{array}{c} \mathcal{N}(u_1[\,(x. := t.)^n\,]) \\ \vdots \\ \mathcal{N}(u_m[\,(x. := t.)^n\,]) \end{array}\right\} \\
&= \mathcal{N}(c\ u[\,(x. := t.)^n\,]^m) \\
&= \mathcal{N}(t[\,(x. := t.)^n\,])
\end{aligned}$$

3. $t = h\ u^m$ where $h \in \mathit{Function} \cup \mathit{Matcher}$: Then

$$\begin{aligned}
\mathcal{N}(t) + \max\{\,(\mathcal{N}(t.))^n\,\} &= 1 + \max\{\,(\mathcal{N}(u.))^m\,\} + \max\{\,(\mathcal{N}(t.))^n\,\} \\
&= 1 + \max\left\{\begin{array}{c} \mathcal{N}(u_1) + \max\{\,(\mathcal{N}(t.))^n\,\} \\ \vdots \\ \mathcal{N}(u_m) + \max\{\,(\mathcal{N}(t.))^n\,\} \end{array}\right\} \\
\text{(by induction hypothesis)} \quad &\geq 1 + \max\left\{\begin{array}{c} \mathcal{N}(u_1[\,(x. := t.)^n\,]) \\ \vdots \\ \mathcal{N}(u_m[\,(x. := t.)^n\,]) \end{array}\right\} \\
&= \mathcal{N}(h\ u[\,(x. := t.)^n\,]^m) \\
&= \mathcal{N}(t[\,(x. := t.)^n\,])
\end{aligned}$$

□

Proposition 2 $\forall t \in \mathit{Term}, q \in \mathit{Program}^- \left(t \xrightarrow[\text{deforest}(q)]{} t' \Rightarrow \mathcal{N}(t) \geq \mathcal{N}(t') \right)$.

PROOF. By induction on the derivation of $t \xrightarrow[\text{deforest}(q)]{} t'$.

1. $t = c\ t^n \xrightarrow[\text{deforest}(q)]{} t_i = t'$ for some $i \in \{1 \ldots n\}$. Rule INDEP must have been used, and then

$$\mathcal{N}(t) = \mathcal{N}(c\ t^n) = \max\{\mathcal{N}(t_1) \ldots \mathcal{N}(t_n)\} \geq \mathcal{N}(t_i) = \mathcal{N}(t') \ .$$

2. $t \xrightarrow[\mathsf{deforest}(q)]{} t'$ where $t \xrightarrow[\mathsf{inner}(q)]{} t'$ by rule OUTIN. I proceed by showing that for all $t \in$ *Term*, if $t \xrightarrow[\mathsf{inner}(q)]{} t'$ then $\mathscr{N}(t) \geq \mathscr{N}(t')$. I will do this by induction on the derivation of $t \xrightarrow[\mathsf{inner}(q)]{} t'$.

   (a) $t = f\ t^n \xrightarrow[\mathsf{inner}(q)]{} t_0[(x.:=t.)^n] = t'$ where $f\ x^n \overset{q}{=} t_0 \in q$. Rule FCALL must have been used. Then

   $$\begin{aligned}
   \mathscr{N}(t) &= \mathscr{N}(f\ t^n) \\
   &= 1 + \max\{\mathscr{N}(t_1) \ldots \mathscr{N}(t_n)\} \\
   &\geq \mathscr{N}(t_0) + \max\{\mathscr{N}(t_1) \ldots \mathscr{N}(t_n)\} \qquad \text{(since } t_0 \text{ is treeless)} \\
   &\geq \mathscr{N}(t_0[(x.:=t.)^n]) \qquad \text{(by Lemma 9)} \\
   &= \mathscr{N}(t')
   \end{aligned}$$

   (b) $t = g\,(c\ t^m)\ t_{m+1} \ldots t_n \xrightarrow[\mathsf{outer}(q)]{} t_0[(x.:=t.)^n] = t'$ where $g\,(c\ x^m)\ x_{m+1} \ldots x_n \overset{q}{=} t_0$. Rule MATCH must have been used. Then

   $$\begin{aligned}
   \mathscr{N}(t) &= \mathscr{N}(g\,(c\ t^m)\ t_{m+1} \ldots t_n) \\
   &= 1 + \max\{\mathscr{N}(c\ t^m)\ \mathscr{N}(t_{m+1}) \ldots \mathscr{N}(t_n)\} \\
   &= 1 + \max\{\max\{\mathscr{N}(t_1) \ldots \mathscr{N}(t_m)\}\ \mathscr{N}(t_{m+1}) \ldots \mathscr{N}(t_n)\} \\
   &= 1 + \max\{\mathscr{N}(t_1) \ldots \mathscr{N}(t_m)\ \mathscr{N}(t_{m+1}) \ldots \mathscr{N}(t_n)\} \\
   &\geq \mathscr{N}(t_0) + \max\{\mathscr{N}(t_1) \ldots \mathscr{N}(t_n)\} \qquad (t_0 \text{ is treeless}) \\
   &\geq \mathscr{N}(t_0[(x.:=t.)^n]) \qquad \text{(by Lemma 9)} \\
   &= \mathscr{N}(t')
   \end{aligned}$$

   (c) $t = g\ x\ t_{n+1} \ldots t_m \xrightarrow[\mathsf{outer}(q)]{} t_0[(x_{n+1}:=t_{n+1}) \ldots (x_m:=t_m)] = t'$ where $g\,(c\ x^n)\ x_{n+1} \ldots x_m \overset{q}{=} t_0$. Rule SPEC must have been used. Then

   $$\begin{aligned}
   \mathscr{N}(t) &= \mathscr{N}(g\ x\ t_{n+1} \ldots t_m) \\
   &= 1 + \max\{\mathscr{N}(x)\ \mathscr{N}(t_{n+1}) \ldots \mathscr{N}(t_m)\} \\
   &= 1 + \max\{\mathscr{N}(t_{n+1}) \ldots \mathscr{N}(t_m)\} \\
   &\geq \mathscr{N}(t_0) + \max\{\mathscr{N}(t_{n+1}) \ldots \mathscr{N}(t_m)\} \qquad (t_0 \text{ is treeless}) \\
   &\geq \mathscr{N}(t_0[(x_{n+1}:=t_1) \ldots (x_{n+m}:=t_m)]) \qquad \text{(by Lemma 9)} \\
   &= \mathscr{N}(t')
   \end{aligned}$$

(d) $t = g\ t_0\ t^n \xrightarrow[\mathsf{outer}(q)]{} g\ t_0'\ t^n = t'$. Rule GDIVE must have been used, so $t_0 \xrightarrow[\mathsf{inner}(q)]{} t_0'$. Then

$$\begin{aligned}
\mathscr{N}(t) &= \mathscr{N}(g\ t_0\ t^n) \\
&= 1 + \max\{\mathscr{N}(t_0)\ \mathscr{N}(t_1) \ldots \mathscr{N}(t_n)\} \\
&\geq 1 + \max\{\mathscr{N}(t_0')\ \mathscr{N}(t_1) \ldots \mathscr{N}(t_n)\} \qquad \text{(by ind. hyp.)} \\
&= \mathscr{N}(g\ t_0'\ t^n) \\
&= \mathscr{N}(t')
\end{aligned}$$

□

I now proceed to show that, for a very restricted class of treeless programs, deforestation will never produce terms with more than one constructor.

Definition 26 (Restricted treeless) Define the subsets $\mathit{Term}^{\neg c}$, $\mathit{Term}^{!c}$, and $\mathit{Term}^{?c}$ of Term as follows.

1. $\mathit{Term}^{\neg c}$ is defined by the grammar

$$\mathit{Term}^{\neg c} \ni t ::= x \mid f\ t^n \mid g\ t^n \qquad \text{(no constructors)}$$

2. $\mathit{Term}^{!c}$ is defined by the grammar

$$\mathit{Term}^{!c} \ni t ::= c\ t^n \mid g\ t\ t^n \qquad \left(\begin{array}{c}\text{single constructor,}\\ \text{possibly under some}\\ \text{pattern functions}\end{array}\right)$$

where $g\,(c\ x^m)\ x_{m+1} \ldots x_n \overset{q}{=} t_0$.

3. $\mathit{Term}^{?c} = \mathit{Term}^{\neg c} \cup \mathit{Term}^{!c}$.

Also, define *restricted treeless terms, definitions and programs* $\mathit{Term}^{\$}$ and $\mathit{Program}^{\$}$ by

$$\begin{array}{lllll}
\mathit{Program}^{\$} & \ni & q & ::= & d^n \\
 & & d & ::= & f\ x^n \overset{q}{=} t \\
 & & & \mid & (g\ p.\ x^n \overset{q}{=} t.)^m \\
\mathit{Term}^{\$} & \ni & t & ::= & r \mid c\ r^n \qquad \text{(at most one constructor)} \\
 & & r & ::= & x \mid f\ x^n \mid g\ x^n
\end{array}$$

□

Proposition 3 $\forall t \in \mathit{Term}^{?c}, q \in \mathit{Program}^{\$} \left(t \xrightarrow[\mathsf{deforest}(q)]{} t' \Rightarrow t' \in \mathit{Term}^{?c} \right)$.

PROOF. By induction on $t \xrightarrow[\mathsf{deforest}(q)]{} t'$:

1. $t = c\ t^n \xrightarrow[\mathsf{deforest}(q)]{} t_i = t'$ for some $i \in \{1 \ldots n\}$ by rule INDEP. Then $t \in \mathit{Term}^{!c}$ and thus $t^n \in \mathit{Term}^{\neg c} \subseteq \mathit{Term}^{?c}$.

2. $t \xrightarrow[\mathsf{deforest}(q)]{} t'$ by rule ALTIN, and then $t \xrightarrow[\mathsf{inner}(q)]{} t'$. I proceed by induction of the derivation of $t \xrightarrow[\mathsf{inner}(q)]{} t'$: for any $t \in \mathit{Term}^{?c}$ and $q \in \mathit{Program}^{\$}$, if $t \xrightarrow[\mathsf{inner}(q)]{} t'$ then $t' \in \mathit{Term}^{?c}$.

   (a) $t = f\ t^n \xrightarrow[\mathsf{inner}(q)]{} t_0[(x.{:=}t.)^n] = t'$ by rule FCALL where $f\ x^n \overset{q}{=} t_0$. Then $t, t^n \in \mathit{Term}^{\neg c}$. Since $t_0 \in \mathit{Term}^{\$} \subseteq \mathit{Term}^{?c}$, it follows easily that $t_0[(x.{:=}t.)^n] \in \mathit{Term}^{?c}$.

   (b) $t = g\,(c\ t^m)\ t_{m+1} \ldots t_n \xrightarrow[\mathsf{inner}(q)]{} t_0[(x.{:=}t.)^n] = t'$ by rule MATCH where $g\,(c\ x^m)\ x_{m+1} \ldots x_n \overset{q}{=} t_0$. Then $t \in \mathit{Term}^{!c}$, so $t^n \in \mathit{Term}^{\neg c}$ and $t_0 \in \mathit{Term}^{\$} \subseteq \mathit{Term}^{?c}$. Proceed as in case (2a).

   (c) $t = g\ x\ t_{m+1} \ldots t_n \xrightarrow[\mathsf{inner}(q)]{} t_0[(x_{m+1}{:=}t_{m+1}) \ldots (x_n{:=}t_n)] = t'$ by rule SPEC where $g\,(c\ x^m)\ x_{m+1} \ldots x_n \overset{q}{=} t_0$. Then $t \in \mathit{Term}^{\neg c}$, so $t^n \in \mathit{Term}^{\neg c}$ and $t_0 \in \mathit{Term}^{\$} \subseteq \mathit{Term}^{?c}$. Proceed as in case (2a).

   (d) $t = g\ t_0\ t^n \xrightarrow[\mathsf{outer}(q)]{} g\ t_0'\ t^n = t'$ by rule GDIVE, and thus $t_0 \xrightarrow[\mathsf{inner}(q)]{} t_0'$. Either both $t, t_0 \in \mathit{Term}^{!c}$ or $t, t_0 \in \mathit{Term}^{\neg c}$. Thus $t_0 \in \mathit{Term}^{!c} \cup \mathit{Term}^{\neg c} = \mathit{Term}^{?c}$, so by the induction hypothesis $t_0' \in \mathit{Term}^{?c} = \mathit{Term}^{!c} \cup \mathit{Term}^{\neg c}$. Also, $t^n \in \mathit{Term}^{\neg c}$. Therefore $t' = g\ t_0'\ t^n \in \mathit{Term}^{?c}$. □

Corollary 1 $\forall t \in \mathit{Term}^{?c}, q \in \mathit{Program}^{\$} : \{t' \mid t \xrightarrow[\mathsf{dalt}(q)]{*} t'\}$ *is finite modulo renaming.*

PROOF. By Proposition 2, there exists $k \in \mathbb{N}$ such that for all $t' \in S : \mathcal{N}(t') \leq k$, which means that there is an upper bound on the number of function symbols for all t'. Moreover, by Proposition 3, for all $t' \in S$: $t' \in \mathit{Term}^{?c}$, so t' contains at most one constructor. In conclusion, there is an upper bound on the number of symbols in any $t' \in S$, and there are only finitely many different terms modulo renaming of a given size. □

It remains to generalise this result to $t \in \mathit{Term}$ and $q \in \mathit{Program}^{-}$. This is sketched below. I start by defining a set of translations.

Definition 27 Define the following four maps:

$$\boxed{\underline{\bullet} \in \mathit{Term} \Rightarrow \mathit{Term}^{\neg c}}$$

$$\begin{array}{lll}
\underline{c\ t^n} & = & f_c\ \underline{t}^n \quad (q = q +\!\!+\ f_c\ x^n = c\ x^n) \\
\underline{f\ t^n} & = & f\ \underline{t}^n \\
\underline{g\ t^n} & = & g\ \underline{t}^n \\
\underline{x} & = & x
\end{array}$$

$$\boxed{\overline{\bullet} \in \mathit{Term}^{-} \Rightarrow \mathit{Term}^{\$}}$$

$$\begin{array}{lll}
\overline{c\ t^n} & = & c\ (f_1\ x^m)\ldots(f_n\ x^m) \quad \begin{pmatrix} q = q +\!\!+\ f_i\ x^m = \overline{t_i}, \\ \{x^m\} = \bigcup_{i=1}^{n} \mathcal{V}(t_i) \end{pmatrix} \\
\overline{f\ x^n} & = & f\ x^n \\
\overline{g\ x^n} & = & g\ x^n \\
\overline{x} & = & x
\end{array}$$

$$\boxed{\overline{\bullet} \in \mathit{Definition}^{-} \Rightarrow \mathit{Definition}^{\$}}$$

$$\begin{array}{lll}
\overline{f\ x^n \overset{q}{=} t} & = & f\ x^n \overset{q}{=} \overline{t} \\
\overline{g\,(c\ x^m)\ x_{m+1}\ldots x_n \overset{q}{=} t} & = & g\,(c\ x^m)\ x_{m+1}\ldots x_n \overset{q}{=} \overline{t}
\end{array}$$

$$\boxed{\overline{\bullet} \in \mathit{Program}^{-} \Rightarrow \mathit{Program}^{\$}}$$

$$\overline{d^n} \quad = \quad \overline{d}^n$$

□

Given some program $q \in \mathit{Program}^{-}$ and term $t \in \mathit{Term}$, by $\underline{t}$ and $\overline{q}$ I then mean the terms $\underline{t}$ as defined above and the program $\overline{q}$ as defined above, but where I include new functions f_c and f_i. The *heart* of a term $\underline{t} \in \mathit{Term}^{\neg c}$ is the result obtained by unfolding all calls to these functions.

Lemma 10 *Let* $t \in \mathit{Term}$ *and* $q \in \mathit{Program}^{-}$. *If* $\{\, t' \mid t \xrightarrow[\mathsf{deforest}(q)]{*} t' \,\}$ *is not finite modulo renaming, then neither is* $\{\, t' \mid \underline{t} \xrightarrow[\mathsf{deforest}(\overline{q})]{*} t' \,\}$.

PROOF (sketch). Assume that $\{\, t' \mid t \xrightarrow[\mathsf{deforest}(q)]{} t' \,\}$ is not finite modulo renaming. Then there is an infinite sequence

$$t = t_0 \xrightarrow[\mathsf{deforest}(q)]{} t_1 \xrightarrow[\mathsf{deforest}(q)]{} t_2 \xrightarrow[\mathsf{deforest}(q)]{} \cdots$$

such that $\bigcup_{i=1}^{\infty} \{\, t_i \,\}$ is not finite modulo renaming. Now show that

$$\begin{tikzcd}
\mathit{Term} \ni & t \arrow[rr, "\mathsf{dalt}(q)"'] & & t' & \in \mathit{Term} \\
\mathit{Term}^{\neg c} \ni & u \arrow[u, "\text{take heart}"] \arrow[rr, dotted, "\mathsf{dalt}(\overline{q})"'] & + & u' \arrow[u, dotted, "\text{take heart}"'] & \in \mathit{Term}^{\neg c}
\end{tikzcd} \tag{4.1}$$

Then consider $\underline{t}$ whose heart is clearly t and use Equation 4.1 to construct an infinite sequence from $\underline{t}$. □

Corollary 2 $\forall t \in \mathit{Term}, q \in \mathit{Program}^{-} (\{\, t' \mid t \xrightarrow[\mathsf{deforest}(q)]{} t' \,\}$ *is finite modulo renaming*).

PROOF. Assume $\{\, t' \mid t \xrightarrow[\mathsf{dalt}(q)]{*} t' \}$ is infinite. Then by Lemma 10, also $\{\, t' \mid \underline{t} \xrightarrow[\mathsf{deforest}(\overline{q})]{*} t' \}$ is infinite, a contradiction. □

4.6 Further Reading

Wadler introduced deforestation in three papers: [121, 120, 29]. Chin improved the practical usability of Wadler's blazed deforestation by devising a higher-order-removal method [18] and automatically skipping "dangerous" parts of the resulting first-order programs [15]. References to many other papers about traditional deforestation appears in a paper by Seidl and Sørensen [98], who deals with more general methods for ensuring termination of deforestation. The division of the transformation process into *symbolic computation*, *search for regularities*, and *program extraction* is due to Pettorossi and Proietti [84], but it can be seen already in the works of Turchin [115].

Shortcut deforestation was formulated by Gill and others [35] as a practical ways of implementing what was seen as the most practical part of Wadler's deforestation: Elimination of intermediate lists. Their transformation works by requiring that every list-producing function uses a primitive `build` construct to build the list, whereas every list consuming function uses `foldr` to traverse the list; when a consumer and a producer function meet, they apply the `foldr`/`build` rule, which cancels the `build` with the `foldr`. What makes the shortcut approach practical is that most of the list-manipulating functions in the standard library (`foldr`, `map`, `partition`, `all`, `takeWhile`, etc.) can be written exclusively using `build` and `foldr`, and thus the elimination of intermediate lists can take place automatically as long as the programmer processes lists by composing standard library functions. Their technique is limited, however, in that only simple, single-threaded list processing can be handled; for instance, `zip` cannot be handled.

Several improvements of this so-called *deforestation in calculation form* have appeared since then. Launchbury and Sheard improved the shortcut method by formulating *warm fusion* [62], a technique based on repeatedly *fusing* two nested `folds` into one, but where the `folds` were defined for several other algebraic data types besides lists. Takano and Meijer [111] generalised the deforestation in calculation form to arbitrary data structures by formulating cancellation rules in category theory, more specifically as *hylomorphisms*, based on Meijer's *Acid Rain Theorem.* Although very powerful, the problem with this approach is that programs have to be written as compositions of functions expressed in a special hylomorphism-triple notation like

$$\begin{array}{rcl} \mathit{map}\ f & \overset{\text{def}}{=} & [\![\mathit{in}_{\text{List}_\beta} \,,\ \mathit{List}(f, \mathit{id}) \,,\ \mathit{out}_{\text{List}_\alpha}]\!] \\ (+\!\!+\ \ \mathit{ys}) & \overset{\text{def}}{=} & [\![\tau\ \mathit{in}_{\text{List}_\alpha} \,,\ \mathit{id} \,,\ \mathit{out}_{\text{List}_\alpha}]\!] \\ & & \text{where } \tau = \lambda(n \nabla c).([\![n \nabla c \,,\ \mathit{id} \,,\ \mathit{out}_{\text{List}_\alpha}]\!]\ \mathit{ys}) \nabla c \end{array}$$

where *in*, *out* and ∇ are related to the initial algebra and final coalgebra induced by the data type List. Similarly, Chitil [19] have devised a type-based deforestation which can perform very aggressive optimisations by relying on an advanced type-inference system (using e.g., second-order constructs) to perform the deforestation.

From my point of view, the essence of deforestation in calculation form is as follows.

1. Introduce special syntax to express consumption and production of data structures (possibly hidden from the programmer by means of libraries).

2. At compile time, perform partial evaluation on the special syntax, treating the special syntax as implicit binding-annotations for static data, and the rest of the program as dynamic data.

3. After partial evaluation, lift the remaining special syntax to ordinary syntax.

As such, the hylomorphism notation is a small domain-specific language for function composition.

Another approach for eliminating intermediate data structures has been proposed by Kühnemann and others (e.g., [57, 56]) based on expressing functional programs as various forms of *tree transducers*. Their results complement traditional deforestation in that they can optimise programs that traditional deforestation cannot, and vise versa. However, only a small class of programs can be expressed in terms of their tree transducers.

Chapter 5

Positive Supercompilation

In this chapter I present *positive supercompilation*, an algorithm that eliminates multiple traversals of data structures. Positive supercompilation was developed as a means of presenting and studying the essential features of *supercompilation*, originally developed by Turchin for the language Refal. Contrary to the original formulation of supercompilation, positive supercompilation only propagates positive information, which can be represented by substitutions, when a program is unfolded.

You will see that positive supercompilation encompasses deforestation in the sense that the effects achieved by deforestation is also achieved by positive supercompilation, but the converse is not true.

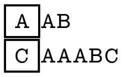
Example 13 Consider the string matching algorithm $\mathit{match}\ p\ s$ that tests whether the list p is contained in the list s:

$$\begin{array}{ll}
\mathit{match}\ p\ s & = m\ p\ s\ p\ s \\
m\ \textsc{nil}\ s\ \mathit{op}\ \mathit{os} & = \textsc{true} \\
m\ (\textsc{cons}\ p\ \mathit{ps})\ s\ \mathit{op}\ \mathit{os} & = m'\ s\ p\ \mathit{ps}\ \mathit{op}\ \mathit{os} \\
m'\ \textsc{nil}\ p\ \mathit{ps}\ \mathit{op}\ \mathit{os} & = \textsc{false} \\
m'\ (\textsc{cons}\ s\ \mathit{ss})\ p\ \mathit{ps}\ \mathit{op}\ \mathit{os} & = \textbf{if}\ p == s\ \textbf{then}\ m\ \mathit{ps}\ \mathit{ss}\ \mathit{os}\ \mathit{op}\ \textbf{else}\ n\ \mathit{op}\ \mathit{os} \\
n\ (\textsc{cons}\ s\ \mathit{ss})\ \mathrm{op} & = m\ \mathit{op}\ \mathit{ss}\ \mathit{op}\ \mathit{ss}
\end{array}$$

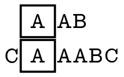
If I were to test whether the pattern $[\,\textsc{a},\textsc{a},\textsc{b}\,]$ is contained in the string

$$[\,\textsc{c},\textsc{a},\textsc{a},\textsc{a},\textsc{b},\textsc{c}\,]\ ,$$

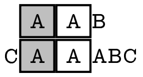
the program would operate as follows. Initially, the first element of the pattern is matched against the first element of the string.



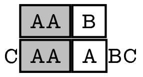
Since there is a mismatch, the pattern cannot match the prefix of the string. The string is therefore shifted one position, and the first element of the pattern is matched with the second element of the string.



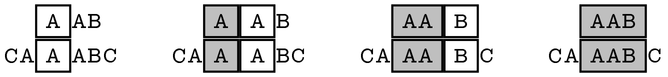
I now have two matching prefixes (which I will indicate by a shade of gray); the next two elements are tested.



Here is also a match, so the program continues with the next two elements.



Alas, a mismatch, which means that the substring starting at the second position of the string does not match the pattern. The program therefore shifts the string one position further, and restarts matching against the original pattern.



The next three steps all match, and thus the matching program returns TRUE.

□

In this chapter I will show how supercompilation can be used to specialise the general-purpose string matcher to a specific pattern, thereby obtaining a more efficient algorithm. For the pattern $[\,\texttt{A}, \texttt{A}, \texttt{B}\,]$, I obtain the program

$$
\begin{array}{lll}
\mathit{main}\ s & = & m_{\mathsf{aab}}\ s \\
m_{\mathsf{aab}}\ [] & = & \texttt{FALSE} \\
m_{\mathsf{aab}}\ (s : ss) & = & \mathbf{if}\ \texttt{A} == s\ \mathbf{then}\ m_{\mathsf{ab}}\ ss\ \mathbf{else}\ m_{\mathsf{aab}}\ ss \\
m_{\mathsf{ab}}\ [] & = & \texttt{FALSE} \\
m_{\mathsf{ab}}\ (s : ss) & = & \mathbf{if}\ \texttt{A} == s\ \mathbf{then}\ m_{\mathsf{b}}\ ss\ \mathbf{else}\ m_{\mathsf{aab}}\ ss \\
m_{\mathsf{b}}\ [] & = & \texttt{FALSE} \\
m_{\mathsf{b}}\ (s : ss) & = & \mathbf{if}\ \texttt{B} == s\ \mathbf{then}\ \texttt{TRUE} \\
 & & \quad \mathbf{else\ if}\ \texttt{A} == s\ \mathbf{then}\ m_{\mathsf{b}}\ ss\ \mathbf{else}\ m_{\mathsf{aab}}\ ss \quad ,
\end{array}
$$

which traverses the string only once.

5.1 Positive Information Propagation

Positive information is propagated by means of substitutions. For instance, consider the non-ground term

$$\mathbf{if}\ x == y\ \mathbf{then}\ \mathrm{t}\ \mathbf{else}\ \mathrm{t}'$$

where t, t' are two (here unspecified) terms. The conditional cannot be directly executed, since I cannot decide whether $x = y$. But *if* x and y evaluates to the same value, I can safely assume that it must hold that any y can be replaced by x in the **then**-branch, i.e., in t. With this observation, I can safely transform $\mathbf{if}\ x == y\ \mathbf{then}\ \mathrm{t}\ \mathbf{else}\ \mathrm{t}'$ into

$$\mathbf{if}\ x == y\ \mathbf{then}\ \mathrm{t}[\,y := x\,]\ \mathbf{else}\ \mathrm{t}' \ .$$

In general, I can have comparisons of the form $\mathrm{t} == \mathrm{t}'$ where both t and t' are not variables. In such cases I can extract a unique most general substitution θ such that $\mathrm{t}\theta = \mathrm{t}'\theta$ provided that the two terms are *unifiable*.

Definition 28 (Unifiers)

1. Two terms $\{\,\mathrm{t}\,,\,\mathrm{t}'\,\} \subseteq \mathit{Term}$ are *disunifiable* if there exists a substitution θ such that $\mathrm{t}\theta \neq \mathrm{t}'\theta$.

2. Two terms $\{\mathrm{t}, \mathrm{t}'\} \subseteq \mathit{Term}$ are *unifiable* if there exists a substitution θ such that $\mathrm{t}\theta = \mathrm{t}'\theta$, and in that case θ is said to be a *unifier for* t and t'. If, for any other unifier θ' for t and t', there exists a substitution σ such that $\theta' = \theta\sigma$, I will say that θ is a *most general unifier* (mgu) of t and t', denoted $\mathrm{t} \sqcup \mathrm{t}'$. □

Lemma 11 *If terms* t *and* t' *are unifiable, then there exists exactly one mgu modulo renaming (of substitutions).*

PROOF. See Lassez, Maher and Marriott [61].

I will now, like in the previous chapter, instrument a semantics of the language so that it can handle non-ground terms, that is, terms that might contain variables. Again, this extension will allow me to produce transformation trace trees from which I can extract new programs. In the literature, the production of such a trace is usually called *driving* the program, and I will stick to this terminology.

Definition 29 (Driving) Let the set of *passive terms* be defined by the grammar

$$\mathit{Passive} \ni \mathrm{a} \ ::= \ \mathrm{x} \mid \mathrm{c}\ \mathrm{a}^{\mathrm{n}}$$

Given a program q, I define the relation $\xrightarrow[\mathsf{posscp(q)}]{} \subseteq \mathit{Term} \times \mathit{Term}$ by the inference system in Figures 5.1 and 5.2. □

The relation $\xrightarrow[\mathsf{posscp(q)}]{}$ is defined in terms of two auxiliary relations, namely $\xrightarrow[\mathsf{pscpout(q)}]{}$ and $\xrightarrow[\mathsf{pscpin(q)}]{}$. In addition to relating terms, these two relations "carry" a substitution with them, denoted by $\xrightarrow[\mathsf{pscpout(q)}]{}_{\theta}$ and $\xrightarrow[\mathsf{pscpin(q)}]{}_{\theta}$. Intuitively, the substitution holds information about what kind of speculative execution has been carried out: The relations $\xrightarrow[\mathsf{pscpin(q)}]{}_{\theta}$ and $\xrightarrow[\mathsf{pscpout(q)}]{}_{\theta}$ seek out a sub-term that can be unfolded, and if such an unfolding is done speculatively, the premises for the performed unfolding are propagated to the $\xrightarrow[\mathsf{posscp(q)}]{}$ relation.

Compare the inference system in Figures 5.1 and 5.2 with the one defined for deforestation (Definition 16). Apart from the inclusion of a if-then-else construct, there are two main differences: The information collected during speculative execution is of a more complex nature, that is, in rule EQUAL the most general unifier for the two terms is calculated and propagated in form of

$$t \in \textit{Term}, x \in \textit{Variable}, c \in \textit{Constructor}, f \in \textit{Function}$$
$$g \in \textit{Matcher}, a \in \textit{Passive}, \theta \in \textit{Substitution}, n \geq 0, m \geq 0$$

$$(\textsc{fcall})\ \frac{f\ x^n \overset{q}{=} t \in q}{f\ t^n \xrightarrow[\mathsf{pscpin}(q)]{}_{[\,]} t[\,(x. := t.)^n\,]}$$

$$(\textsc{match})\ \frac{g\,(c\ x^m)\ x_{m+1} \ldots x_n \overset{q}{=} t \in q}{g\,(c\ t^m)\ t_{m+1} \ldots t_n \xrightarrow[\mathsf{pscpin}(q)]{}_{[\,]} t[\,(x. := t.)^n\,]}$$

$$(\textsc{spec})\ \frac{g\ p\ x^n = t \in q}{g\ x\ t^n \xrightarrow[\mathsf{pscpin}(q)]{}_{[\,x := p\,]} t[\,(x. := t.)^n\,]}$$

$$(\textsc{gdive})\ \frac{t \xrightarrow[\mathsf{pscpin}(q)]{}_{\theta} t' \qquad g\ p\ x^n = t \in q}{g\ t\ t^n \xrightarrow[\mathsf{pscpin}(q)]{}_{\theta} g\ t'\ t^n}$$

$$(\textsc{equal})\ \frac{a \text{ and } a' \text{ are unifiable} \qquad \theta = a \sqcup a'}{\textbf{if } a == a' \textbf{ then } t \textbf{ else } t' \xrightarrow[\mathsf{pscpin}(q)]{}_{\theta} t}$$

$$(\textsc{neq})\ \frac{a \text{ and } a' \text{ are disunifiable}}{\textbf{if } a == a' \textbf{ then } t \textbf{ else } t' \xrightarrow[\mathsf{pscpin}(q)]{}_{[\,]} t'}$$

$$(\textsc{ifdive})\ \frac{t_1 \xrightarrow[\mathsf{pscpout}(q)]{}_{\theta} t_1'}{\textbf{if } t_1 == t_2 \textbf{ then } t_3 \textbf{ else } t_4 \xrightarrow[\mathsf{pscpin}(q)]{}_{\theta} \textbf{if } t_1' == t_2 \textbf{ then } t_3 \textbf{ else } t_4}$$

$$(\textsc{ifdive'})\ \frac{t_2 \xrightarrow[\mathsf{pscpout}(q)]{}_{\theta} t_2'}{\textbf{if } a == t_2 \textbf{ then } t_3 \textbf{ else } t_4 \xrightarrow[\mathsf{pscpin}(q)]{}_{\theta} \textbf{if } a == t_2' \textbf{ then } t_3 \textbf{ else } t_4}$$

Figure 5.1: Inner unfold rules for positive supercompilation. The positive information is collected from the strict comparison operation in conditionals, and carried around in a substitution.

$$t \in \textit{Term}, c \in \textit{Constructor}, a \in \textit{Passive}, \theta \in \textit{Substitution}, n \geq 0, m \geq 0$$

$$(\textsc{outin})\ \frac{t \xrightarrow[\mathsf{pscpin}(q)]{}_{\theta} t'}{t \xrightarrow[\mathsf{pscpout}(q)]{}_{\theta} t'}$$

$$(\textsc{cdive})\ \frac{t \xrightarrow[\mathsf{pscpout}(q)]{}_{\theta} t'}{c\ a^{m-1}\ t\ t_{m+1} \ldots t_n \xrightarrow[\mathsf{pscpout}(q)]{}_{\theta} c\ a^{m-1}\ t'\ t_{m+1} \ldots t_n}$$

$$(\textsc{prop})\ \frac{t \xrightarrow[\mathsf{pscpout}(q)]{}_{\theta} t' \qquad \theta \text{ is free for } t}{t \xrightarrow[\mathsf{posscp}(q)]{} t'\theta}$$

Figure 5.2: Outer unfold rules for positive supercompilation. Positive information collected in substitutions is propagated into the whole term.

a substitution. Furthermore, every substitution collected by the unfolding of a sub-term is propagated to the *surrounding* term by rule PROP. You will see how this kind of propagation increases both the strength of the transformation and the danger of non-termination.

As in the previous chapter, I will now define the notion of a *transformation trace* on top of $\xrightarrow[\mathsf{posscp}(q)]{}$.

Definition 30 (Transformation trace) Given a program q and a term t, define the *transformation trace of* t as $\Delta_t^{\xrightarrow{\mathsf{posscp}(q)}}$, with the following requirements.

1. If $t = g\ x\ t_{n+1} \ldots t_m$, then I assume that the children appear in the order corresponding to the patterns defining g.
2. If $t = \textbf{if } a_1 == a_2 \textbf{ then } t_3 \textbf{ else } t_4$, then I assume that the children appear in the order t_3, t_4. □

Like with deforestation, I can now establish a *model* of a particular program by generating a transformation trace that will portray all possible computation traces.

Example 14 Consider a program for checking that a list is a prefix of another list.

$$\begin{array}{lcl} \mathit{pre}\ []\ \mathit{ys} & = & \textsc{true} \\ \mathit{pre}\ (x : \mathit{xs})\ \mathit{ys} & = & h\ \mathit{ys}\ \mathit{xs}\ x \\ h\ []\ \mathit{xs}\ x & = & \textsc{true} \\ h\ (y : \mathit{ys})\ \mathit{xs}\ x & = & \textbf{if}\ x == y\ \textbf{then}\ \mathit{pre}\ \mathit{xs}\ \mathit{ys}\ \textbf{else}\ \textsc{false}\ . \end{array}$$

If we assume that PAIR is a binary constructor, driving the term

$$\mathit{pre}\ [\,\textsc{pair}\ x\ x, x\,]\ [\,\textsc{pair}\ y\ y, y, y\,]$$

w.r.t. the above program gives us the transformation trace shown in Figure 5.3. Notice that $(\textsc{pair}\ x\ x) \sqcup (\textsc{pair}\ y\ y) = [\,x := y\,]$, and further how this substitution is applied to the rest of the term so that

$$\begin{array}{l} \textbf{if}\ (\textsc{pair}\ x\ x) == (\textsc{pair}\ y\ y) \\ \textbf{then}\ \mathit{pre}\ [\,x\,]\ [\,y, y\,]\ \textbf{else}\ \textsc{false} \end{array} \quad \xrightarrow[\mathsf{posscp(q)}]{} \quad \mathit{pre}\ [\,y\,]\ [\,y, y\,]\ .$$

□

In general, of course, driving a program is not guaranteed to terminate. As a first approximation to a terminating algorithm for positive supercompilation, I can reuse Definition 19 (page 58) to prune the transformation trace tree. I can also modify the program extraction algorithm from Definition 21 (page 59) to account for conditionals.

Definition 31 (Decomposition) If a term t is not passive, it can be decomposed as e[r] where e and r are defined by the grammars below and e[r] denotes replacing the hole • in e by r.

$$\begin{array}{lcl@{\qquad}lcl} e & ::= & \bullet & r & ::= & f\ t^n \\ & | & c\ a^n\ e\ t^m & & | & g\ o\ t^n \\ & | & g\ e\ t^n & & | & \textbf{if}\ a == a'\ \textbf{then}\ t\ \textbf{else}\ t' \\ & | & \textbf{if}\ e == t\ \textbf{then}\ t'\ \textbf{else}\ t'' & o & ::= & x \\ & | & \textbf{if}\ a == e\ \textbf{then}\ t'\ \textbf{else}\ t'' & & | & c\ t^n\ . \end{array}$$

□

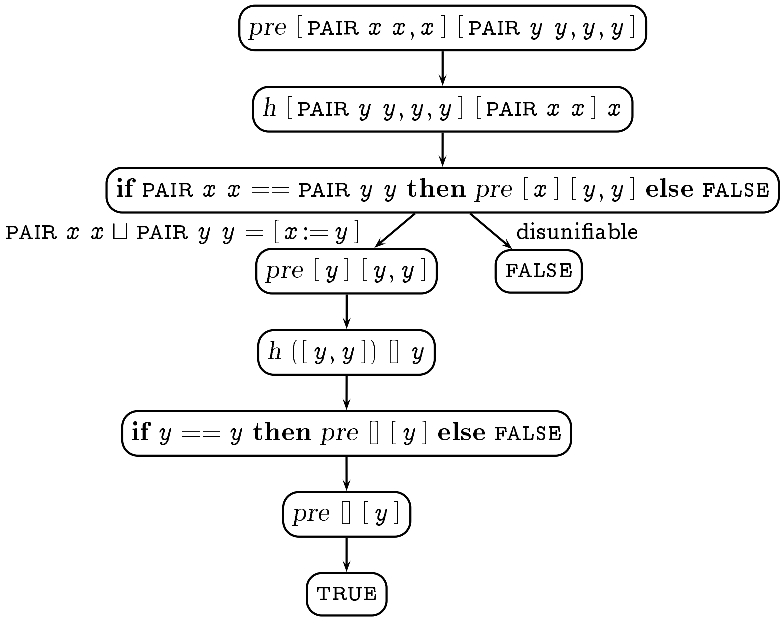

pre [PAIR x x, x] [PAIR y y, y, y]
h [PAIR y y, y, y] [PAIR x x] x
if PAIR x x $==$ PAIR y y **then** pre [x] [y, y] **else** FALSE
PAIR x $x \sqcup$ PAIR y $y = [x := y]$
disunifiable
pre [y] [y, y]
FALSE
h ([y, y]) [] y
if $y == y$ **then** pre [] [y] **else** FALSE
pre [] [y]
TRUE

Figure 5.3: Transformation trace using positive information. The left branch uses the information that the unifier of PAIR x x and PAIR y y is $[x := y]$.

```
cogen $\tau\ \eta$ =
  let $\langle x^n \rangle = \mathscr{V}(\tau\eta)$
  in if $\eta \in \text{leaves}_\tau$
     then if $\tau\eta \in \mathit{Passive}$
          then $\langle \ulcorner \tau\eta \urcorner , \varnothing \rangle$
          else let $\mu$ suchthat $\mu <_\tau \eta \wedge \tau\mu \equiv \tau\eta$
               in $\langle \ulcorner h_\mu\ x^n \urcorner , \varnothing \rangle$
     else let $\langle \langle \ulcorner t_. \urcorner , q_. \rangle^m \rangle$ = map (cogen $\tau$) (children$_\tau \eta$)
          in if $\tau\eta = \ulcorner e[g\ x \ldots] \urcorner$ (for some $e$)
             then $\left\langle \ulcorner g_\eta\ x^n \urcorner , \bigcup_{i \in \{0 \ldots m\}} q_i \right\rangle$
                  where $q_0$ = {
                       $\ulcorner g_\eta\ (c_1\ y^k)\ x_2\ \ldots\ x_n = t_1 \urcorner$
                            where $\langle y^k \rangle = \mathscr{V} t_1 \dotdiv \mathscr{V}(\tau\eta)$
                                      $\vdots$
                       $\ulcorner g_\eta\ (c_m\ y^k)\ x_2\ \ldots\ x_n = t_1 \urcorner$
                            where $\langle y^k \rangle = \mathscr{V} t_m \dotdiv \mathscr{V}(\tau\eta)$
                       where $c^m$ are the constructors from
                                  the definition of $g$
                  }
             else if $\tau\eta = \ulcorner e[$if $a == a'$ then $t$ else $t'] \urcorner$ (for some $e$) $\wedge\ m = 2$
             then $\langle \ulcorner f_\eta\ x^n \urcorner , \{ \ulcorner f_\eta\ x^n =$ if $a == a'$ then $t_1$ else $t_2 \urcorner \} \cup q_1 \cup q_2 \rangle$
             else $\langle \ulcorner f_\eta\ x^n \urcorner , \{ \ulcorner f_\eta\ x^n = t_1 \urcorner \} \cup q_1 \rangle$
```

Figure 5.4: Extracting a program from a finite transformation trace.

Definition 32 (Program extraction) Given a finite transformation trace τ, the program extracted from τ is

$$\{\ulcorner \mathit{main}\ x^n = t \urcorner\} \cup q \quad \textbf{where} \begin{array}{l} \langle t , q \rangle := \text{cogen}\ \tau\ \langle\rangle \\ \langle x^n \rangle = \mathscr{V} t \ , \end{array}$$

where 'cogen' is defined by the algorithm in Figure 5.4. □

Compared to the program extraction I devised for deforestation in Chapter 4, conditionals are accounted for by producing a conditional in the transformed program only if the original conditional gave rise to *two* branches in the transformation trace tree. The rationale behind this is that, if the original conditional only gave rise to *one* branch, the truth value of the condition could be decided during driving, and there is thus no need for a conditional in the transformed program, similar to what happens when driving a term of the form ‘$\mathtt{e[g\ (c\ t^{n})\ \ldots]}$’. A preliminary version of positive SCP can now be formulated as follows.

Definition 33 (Preliminary positive SCP) Given $\mathsf{q} \in$ *Program* and $\mathsf{t} \in$ *Term*.

1. Construct the transformation trace $\tau = \Delta_{\mathsf{q}}^{\overrightarrow{\mathsf{posscp(q)}}}$.
2. Prune τ by using Definition 19 to get a finite tree τ'.
3. Construct a new program from τ' by using Definition 32. □

Comparing deforestation with this preliminary version of positive supercompilation, you can see that the effects achieved by deforestation can also be achieved by supercompilation. In fact, the clever way information is propagated during driving enables the transformer to perform even further optimisations. Let me give an example of this.

Example 15 Consider again the string matching program. As promised at the beginning of this chapter, positive supercompilation can be used to specialise the matcher to a specific pattern, say $[\mathrm{A},\mathrm{A},\mathrm{B}]$. The transformation trace tree produced is depicted in Figure 5.5. The program extracted from this tree is, with some unnecessary functions removed,

$$\begin{aligned}
&m_{\mathsf{aab}}\ [] && = \mathrm{FALSE}\\
&m_{\mathsf{aab}}\ (s : ss) && = h_{\mathsf{aab}}\ s\ ss\\
&h_{\mathsf{aab}}\ s\ ss && = \mathbf{if}\ \mathrm{A} == s\ \mathbf{then}\ m_{\mathsf{ab}}\ ss\ \mathbf{else}\ m_{\mathsf{aab}}\ ss\\
&m_{\mathsf{ab}}\ [] && = \mathrm{FALSE}\\
&m_{\mathsf{ab}}\ (s : ss) && = h_{\mathsf{ab}}\ s\ ss\\
&h_{\mathsf{ab}}\ s\ ss && = \mathbf{if}\ \mathrm{A} == s\ \mathbf{then}\ m_{\mathsf{b}}\ ss\ \mathbf{else}\ h_{\mathsf{aab}}\ s\ ss\\
&m_{\mathsf{b}}\ [] && = \mathrm{FALSE}\\
&m_{\mathsf{b}}\ (s : ss) && = h_{\mathsf{b}}\ s\ ss\\
&h_{\mathsf{b}}\ s\ ss && = \mathbf{if}\ \mathrm{B} == s\ \mathbf{then}\ \mathrm{TRUE}\ \mathbf{else}\ h_{\mathsf{ab}}\ s\ ss\ .
\end{aligned}$$

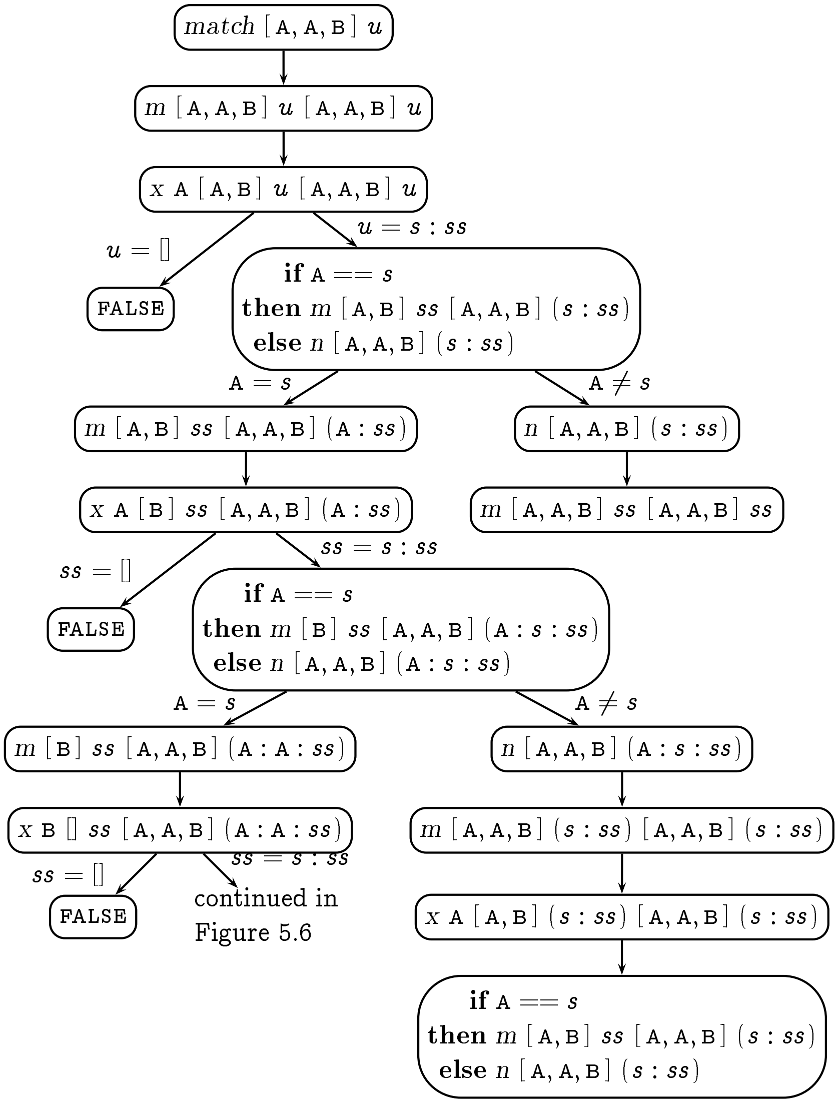

match [A, A, B] u
m [A, A, B] u [A, A, B] u
x A [A, B] u [A, A, B] u
u = []
u = s : ss
FALSE
if A == s
then m [A, B] ss [A, A, B] (s : ss)
else n [A, A, B] (s : ss)
A = s
A ≠ s
m [A, B] ss [A, A, B] (A : ss)
n [A, A, B] (s : ss)
x A [B] ss [A, A, B] (A : ss)
m [A, A, B] ss [A, A, B] ss
ss = []
ss = s : ss
FALSE
if A == s
then m [B] ss [A, A, B] (A : s : ss)
else n [A, A, B] (A : s : ss)
A = s
A ≠ s
m [B] ss [A, A, B] (A : A : ss)
n [A, A, B] (A : s : ss)
x B [] ss [A, A, B] (A : A : ss)
m [A, A, B] (s : ss) [A, A, B] (s : ss)
ss = []
ss = s : ss
FALSE
continued in
Figure 5.6
x A [A, B] (s : ss) [A, A, B] (s : ss)
if A == s
then m [A, B] ss [A, A, B] (s : ss)
else n [A, A, B] (s : ss)

Figure 5.5: Driving the string matcher with the pattern [A, A, B]

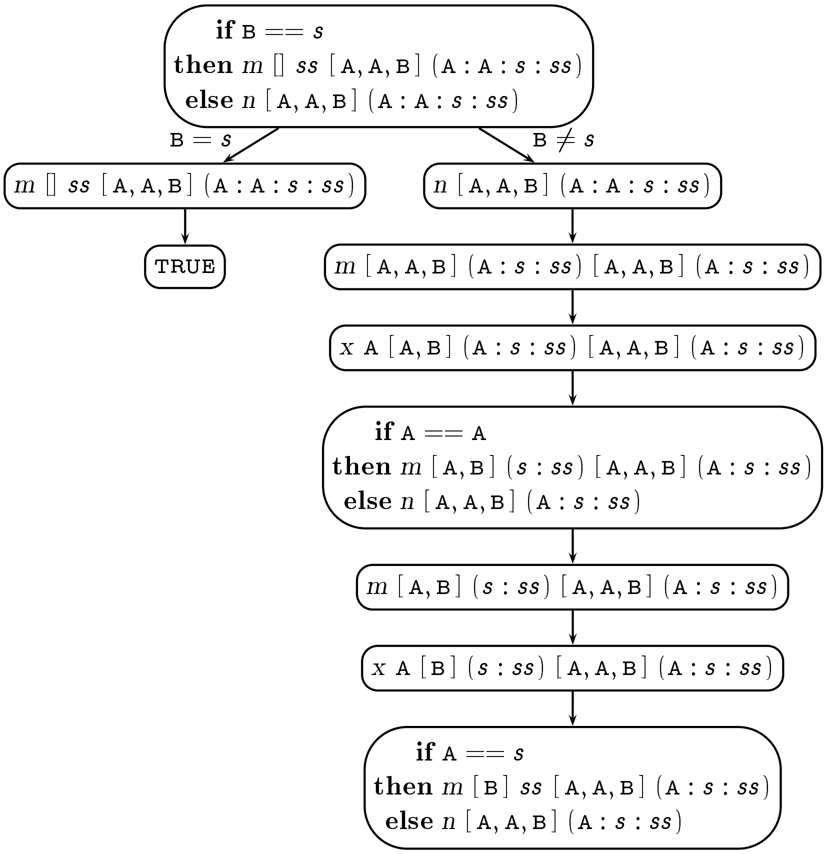
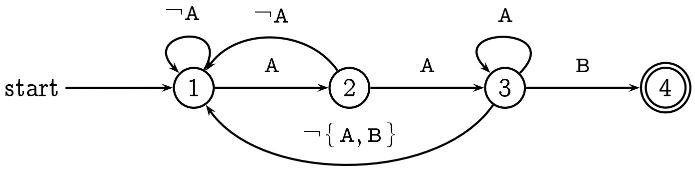

if B == s
then m [] ss [A, A, B] (A : A : s : ss)
else n [A, A, B] (A : A : s : ss)
B = s
B ≠ s
m [] ss [A, A, B] (A : A : s : ss)
n [A, A, B] (A : A : s : ss)
TRUE
m [A, A, B] (A : s : ss) [A, A, B] (A : s : ss)
x A [A, B] (A : s : ss) [A, A, B] (A : s : ss)
if A == A
then m [A, B] (s : ss) [A, A, B] (A : s : ss)
else n [A, A, B] (A : s : ss)
m [A, B] (s : ss) [A, A, B] (A : s : ss)
x A [B] (s : ss) [A, A, B] (A : s : ss)
if A == s
then m [B] ss [A, A, B] (A : s : ss)
else n [A, A, B] (A : s : ss)

Figure 5.6: Driving the string matcher (cont.)

The transformed program is in fact a *finite!state machine*, which graphically can be depicted thus:



However, the transformed program is not optimal because, if the test $\texttt{A} == s$ fails in h_{ab}, then the same test is performed once again in h_{aab}. This unnecessary second test is performed because no *negative* information is propagated. I will remedy this situation in the next chapter. □

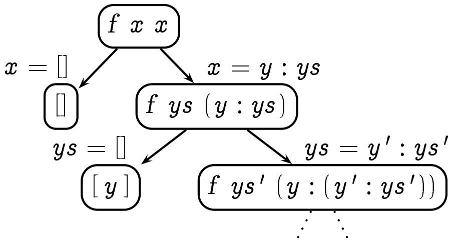
5.2 When to Stop

The preceding algorithm does not always terminate. As in the case of deforestation, I can summarise the causes of non-termination into a few canonical examples. It turns out that in addition to *the accumulating parameter* and *the obstructing function call*, only one additional example is needed to characterise non-termination for positive supercompilation, namely *the accumulating side effect.*

Example 16 Consider the program

$$\begin{aligned} &f\ []\ zs &&= zs \\ &f\ (y : ys)\ zs &&= f\ ys\ zs\quad . \end{aligned}$$

Driving the term '$f\ x\ x$' w.r.t. the above program will give the infinite transformation tree



The problem is that the positive information derived from speculatively executing the second definition of *f* will make the second parameter grow unboundedly.

□

In all three canonical cases, the problem of the growing terms can collectively be described in terms of the *homeomorphic embedding* relation (slightly modified to treat variables).

Definition 34 (Homeomorphic embedding) The homeomorphic embedding relation $\trianglelefteq_{\text{hom}} \subseteq \mathit{Term} \times \mathit{Term}$ is the smallest relation on terms satisfying

$$\frac{}{x \trianglelefteq_{\text{hom}} y} \qquad \frac{\exists i \in \{1 \ldots n\} \; (\, t \trianglelefteq_{\text{hom}} t_i \,)}{t \trianglelefteq_{\text{hom}} h \; t^n} \qquad \frac{\forall i \in \{1 \ldots n\} \; (\, t_i \trianglelefteq_{\text{hom}} t'_i \,)}{h \; t^n \trianglelefteq_{\text{hom}} h \; t'^n} \; ,$$

where $\{ \, x \, , \, y \, \} \subseteq \mathit{Variable}$, $t \in \mathit{Term}$ and $h \in \mathit{Constructor} \cup \mathit{Function} \cup \mathit{Matcher} \cup \{ \, \textbf{ifthenelse} \, \}$. □

The first rule says that a variable is embedded in a variable. The second says that a term t is embedded in another term t' if it is embedded in one of the subterms of t'. The third rules says a term t is embedded in another term t' if their roots have identical labels and each sub-term of t is embedded in the corresponding sub-term of t'.

Intuitively, a term t is homeomorphic embedded in another term t' if t' contains all parts of t in the right order. Another way of putting it, t is homeomorphic embedded in t' if one can obtain t' from t by repeatedly replacing subterms u in t with terms that contain u.

Example 17 The table below gives examples and non-examples of homeomorphic embedding.

$$\begin{array}{rll@{\qquad\qquad}lll}
c & \trianglelefteq_{\text{hom}} & f \; c & f \; (c \; x) & \not\trianglelefteq_{\text{hom}} & c \; x \\
h \; c & \trianglelefteq_{\text{hom}} & h \; (f \; c) & f \; (c \; x) & \not\trianglelefteq_{\text{hom}} & c \; (f \; x) \\
h \; c \; x & \trianglelefteq_{\text{hom}} & h \; (f \; c) \; (f \; y) & f \; (c \; x) & \not\trianglelefteq_{\text{hom}} & f \; (f \; (f \; x))
\end{array}$$

where $\{ \, x \, , \, y \, \} \subseteq \mathit{Variable}$, $c \in \mathit{Constructor}$, $f \in \mathit{Function}$ and $h \in \mathit{Matcher}$. □

If we now return to inspect the three canonical examples of nontermination, you can see that, in all three cases, there is indeed a subsequence of nodes

$\langle \eta_0 , \eta_1 , \eta_2 , \ldots \rangle$ in the infinite branches of the transformation trace τ such that

$$\tau(\eta_0) \trianglelefteq_{\text{hom}} \tau(\eta_1) \trianglelefteq_{\text{hom}} \tau(\eta_2) \trianglelefteq_{\text{hom}} \cdots .$$

In fact, it turns out that in every infinite branch, there must be a term that *homeomorphically embeds* an ancestor.

Theorem 2 (Kruskal's Tree Theorem) *For any infinite sequence of terms* $\langle t_0 , t_1 , \ldots \rangle$*, there exists* $\{ i , j \} \subseteq \mathbb{N}$ *such that* $i < j$ *and* $t_i \trianglelefteq_{hom} t_j$.

This is very good news, because it means that if I always stop driving a term t' when there is a term t (in an ancestor node) which is homeomorphic embedded in t', I will never be able to produce an infinite branch in the transformation tree; and since every tree is finitely branching, I will only produce finite trees. Moreover, since $t \trianglelefteq_{\text{hom}} t'$, all subexpressions of t are present in t'. This suggests that t' *might* give rise to some infinite sequence of terms, so driving is stopped for a good reason.

I am now faced with a new question: What should I do when the above situation arises?

5.3 Generalisation

The answer is to split up the offending term into separate parts. To keep the transformation trace tree from falling apart, I introduce a new term construct that can hold the separate parts of a term.

Definition 35 (Let-term) The set of *let-terms*, denoted *Term*$^{\text{let}}$, is defined by the grammar

$$\mathit{Term}^{\text{let}} \ni \ell ::= t \mid \textbf{let } (x_. := t_.)^n \textbf{ in } t \ ,$$

where $x_i \neq x_j$ for all $i \neq j$, $x^n \in \mathscr{V}(t)$ and $t \notin \mathit{Variable}$. A **let**-term ℓ is *proper* iff $\ell \notin \mathit{Term}$, and the *heart* of a proper **let**-term **let** $(x_. := t_.)^n$ **in** t is $t[\,(x_. := t_.)^n\,]$. Furthermore, I require that the heart of a **let**-term is not a renaming of its right-hand side, that is, $t \not\equiv t[\,(x_. := t_.)^n\,]$. □

By the above definition I trivially have that $\mathit{Term} \subseteq \mathit{Term}^{\text{let}}$, and thus all the transformation traces I have seen so far are trees over **let**-terms. The intended interpretation of a **let**-term ℓ in a tree is that ℓ should be replaced by its heart when the transformation is over.

The question is now *how* to split up a term t' that embeds another term t. I do this by *generalising* t'. The idea behind *generalisation* is to extract as much common structure as possible of the two terms.

Definition 36 (generalisation)

1. A term t' is an *instance* of a term t, denoted $t' \geq_{\text{inst}} t$, if there exists a substitution θ such that $t\theta = t'$.

2. A generalisation of two terms t and t' is a term u such that $t \geq_{\text{inst}} u$ and $t' \geq_{\text{inst}} u$.

3. A *most specific generalisation* (msg) of two terms t and t' is a generalisation u such that, for all other generalisation u' of t and t', I have that $u \geq_{\text{inst}} u'$. I denote the msg of t and t' by $t \sqcap t$.

4. If $t \sqcap t'$ is a variable, I say that t and t' are *incommensurable*, denoted $t || t'$. □

Example 18 The following table gives examples of the most specific generalisation. Let $\{x, y\} \subseteq$ *Variable* and $\{c, f, h\} \subseteq$ *Constructor* $\cup$ *Function* $\cup$ *Matcher* $\cup \{$ **ifthenelse** $\}$.

t	θ	$t \sqcap t'$	θ'	t'
c	$[x{:=}c]$	x	$[x{:=}f\ c]$	$f\ c$
$h\ c$	$[x{:=}c]$	$h\ x$	$[x{:=}f\ c]$	$h\ (f\ c)$
$h\ x$	$[\,]$	$h\ x$	$[x{:=}f\ x]$	$h\ (f\ x)$
$h\ c\ c$	$[x{:=}c]$	$h\ x\ x$	$[x{:=}f\ c]$	$h\ (f\ c)\ (f\ c)$

□

Generalisation is the dual of unification, which is why it is also called *anti-unification*: A unifier of two terms t and t' is a *substitution* θ that maps both t and t' to some t_u, whereas a generalisation of t and t' is a *term* t_g such that both t and t' are instances of t_g. Graphically,

$$\begin{tikzcd}
& t_u & \\
t \arrow[ur, "\theta"] & & t' \arrow[ul, "\theta"'] \\
& t_g \arrow[ul, "\theta_1"] \arrow[ur, "\theta_2"'] &
\end{tikzcd}$$

Lemma 12 (Unique msg) *For any two terms* $t, u \in$ *Term there exists exactly one msg modulo renaming.*

PROOF. See Lassez, Maher and Marriott [61].

The msg of two terms t and t' and the corresponding substitutions can be obtained by exhaustive application of the following rewrite system to the initial triple $\langle x\,,\,[x{:=}t]\,,\,[x{:=}t']\rangle$:

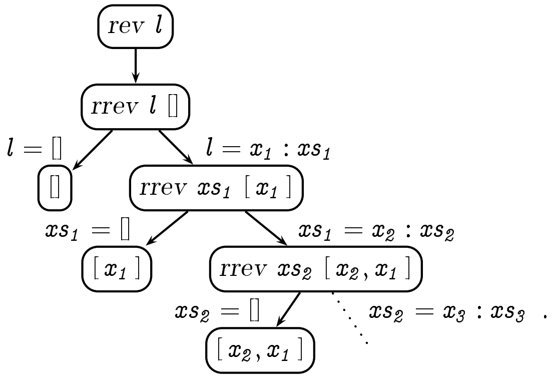
$$\begin{array}{c}
\left\langle \begin{array}{l} t_g \\ [x{:=}h\ t^n] \cup \theta_1 \\ [x{:=}h\ u^n] \cup \theta_2 \end{array} \right\rangle \xrightarrow[\mathsf{msg}]{} \left\langle \begin{array}{l} t_g[x{:=}h\ y^n] \\ [(y.{:=}t.)^n] \cup \theta_1 \\ [(y.{:=}u.)^n] \cup \theta_2 \end{array} \right\rangle \\[2ex]
\left\langle \begin{array}{l} t_g \\ [x{:=}t, y{:=}t] \cup \theta_1 \\ [x{:=}u, y{:=}u] \cup \theta_2 \end{array} \right\rangle \xrightarrow[\mathsf{msg}]{} \left\langle \begin{array}{l} t_g[x{:=}y] \\ [y{:=}t] \cup \theta_1 \\ [y{:=}u] \cup \theta_2 \end{array} \right\rangle
\end{array} \tag{5.1}$$

Now that I have a way to extract common initial structures of two terms, let us see how this can be used to dismantle problematic terms in a transformation tree.

Example 19 (msg on accumulating parameter) Consider again the program

```
rev xs           = rrev xs []
rrev [] ys       = ys
rrev (x : xs) ys = rrev xs (x : ys) .
```

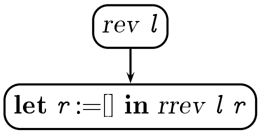
The problem here is that functions *rrev* has an *accumulating parameter* which will grow unboundedly during transformation of $\mathit{rev}\ l$:



The first embedding is the term '$\mathit{rrev}\ l\ [\,]$' which is embedded in the term '$\mathit{rrev}\ xs_1\ [\,x_1\,]$', and I have that

$$(\mathit{rrev}\ l\ [\,]) \sqcap (\mathit{rrev}\ xs_1\ [\,x_1\,])\ =\ \mathit{rrev}\ l\ r$$

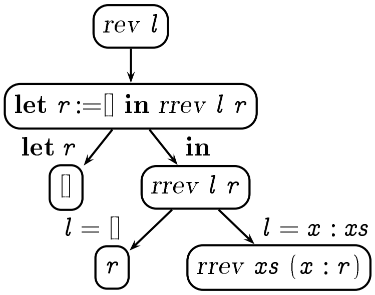
with the substitutions $[\,r := [\,]\,]$ and $[\,l := xs_1\,, r := [\,x_1\,]\,]$.[1] The msg is strictly more general than both the original terms, so I replace the subtree containing both the nodes with a **let**-term built up from the msg, the rationale being that such a term represent both evicted terms:



This kind of operation is called a *generalisation step*. I then continue to drive

[1] For technical reasons, more specifically code generation, I require that the msg will reuse as many variables as possible from the terms.

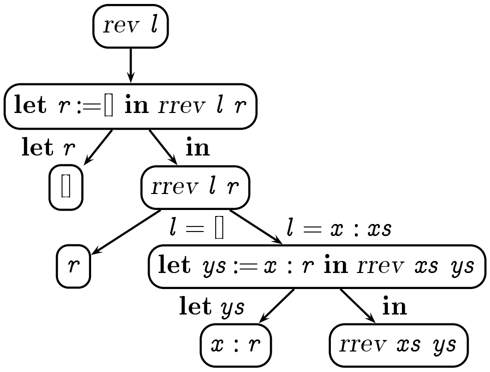
each part of the **let**-term independently:



The new kinds of labels on the edges are used to clarify where a child fits into the parent: "**let** x" means that the sub-tree represents the variable x, and "**in**" means that the subtree represents the right-hand side of the **let**-term. Looking at the tree, I now have $\mathit{rrev}\ l\ r \trianglelefteq_{\text{hom}} \mathit{rrev}\ \mathit{xs}\ (x : r)$, but unfortunately

$$(\mathit{rrev}\ l\ r) \sqcap (\mathit{rrev}\ \mathit{xs}\ (x : r)) \ = \ \mathit{rrev}\ l\ r \ \ ,$$

that is, the larger term is an instance of the smaller. This means that there would be no point in replacing the whole subtree containing both the terms, as I did before, because such a replacement would lead to exactly the same tree. Instead, I restrain myself to only replace the leaf labelled $\mathit{rrev}\ \mathit{xs}\ (x : r)$ with a **let**-term in which the right-hand side is a renaming of '$\mathit{rrev}\ l\ r$', and continue driving:



The rightmost term is now a renaming of an ancestor. I have thus been able to get a finite transformation trace by performing a series of generalisation steps. □

I can now formulate a set of generalisation operations that will be used to ensure termination of positive supercompilation. These operations are summarised in Figures 5.8 and 5.7. The drive operation assumes that the unfold operation is extended to **let**-terms. In view of the above, I can extend driving to accommodate **let**-terms by adding the following inference rule.

$$(\textsc{let})\ \frac{}{\mathbf{let}\ (x_{\cdot} := t_{\cdot})^{n-1}\ \mathbf{in}\ t_n \xrightarrow[\mathsf{posscp}(q)]{} t_i}\ i \in \{1, \ldots, n\}\ .$$

The split operation is needed when the offending terms are incommensurable (i.e., when their msg is a variable). This operation simply turns the label of a node into a **let**-term by peeling off the outermost symbol. This operation is needed because otherwise, by using the msg (which is a variable) to form a **let**-term, I would not break the term up in smaller pieces, and thus no progress would be made.

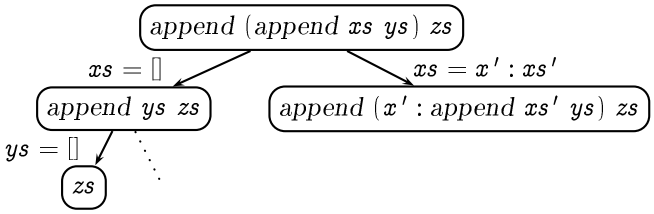
The upwards and downwards generalisations happen when the offending terms have a msg that is not a variable, as illustrated in Example 19.

I now have the raw material for fitting together an algorithm for positive supercompilation. Unfortunately, it turns out that the algorithm will be a bit too eager to stop, when I use the homeomorphic embedding relation as the sole control mechanism on driving.

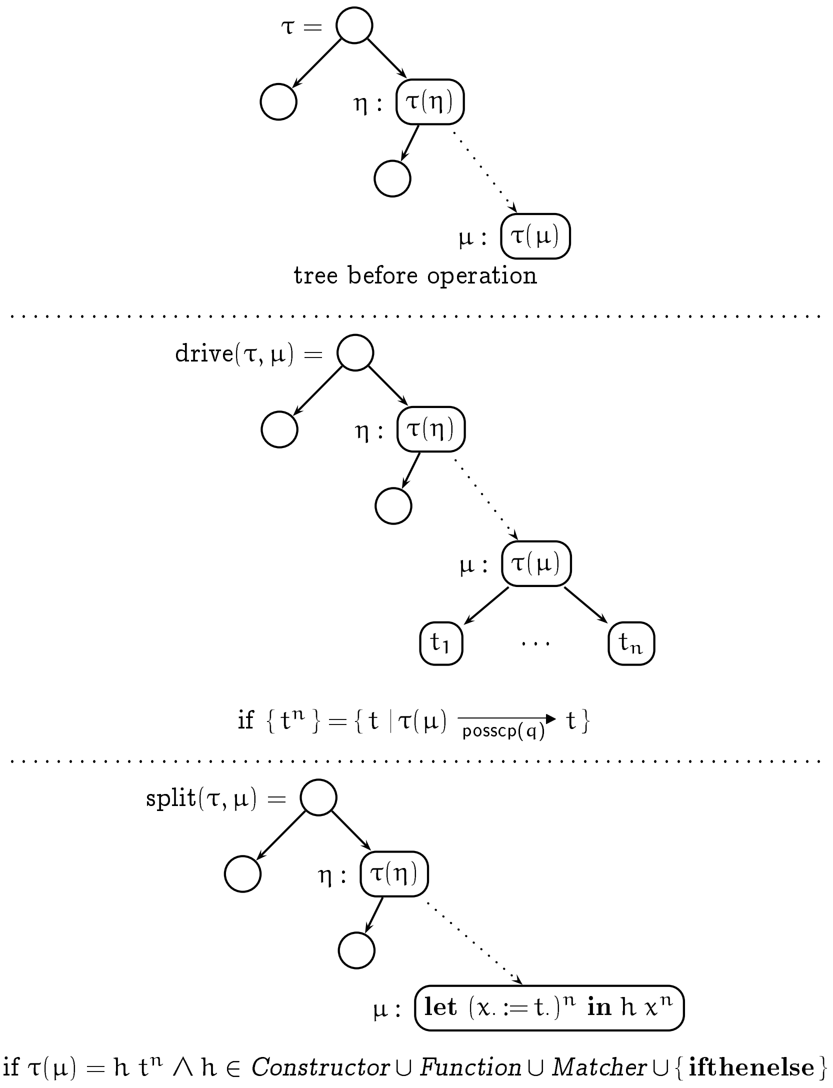
Example 20 ($\trianglelefteq_{\mathbf{hom}}$ too eager) Consider again the append program

$$\begin{aligned} &\mathit{append}\ []\ \mathit{ys} &&= \mathit{ys} \\ &\mathit{append}\ (x : \mathit{xs})\ \mathit{ys} &&= x : \mathit{append}\ \mathit{xs}\ \mathit{ys}\ \ . \end{aligned}$$

The initial transformation trace tree of '*append* (*append xs ys*) *zs*' will look like this:



The root label is embedded in the label of the rightmost node, so, following the rationale above, a generalisation would have to be made. But driving the

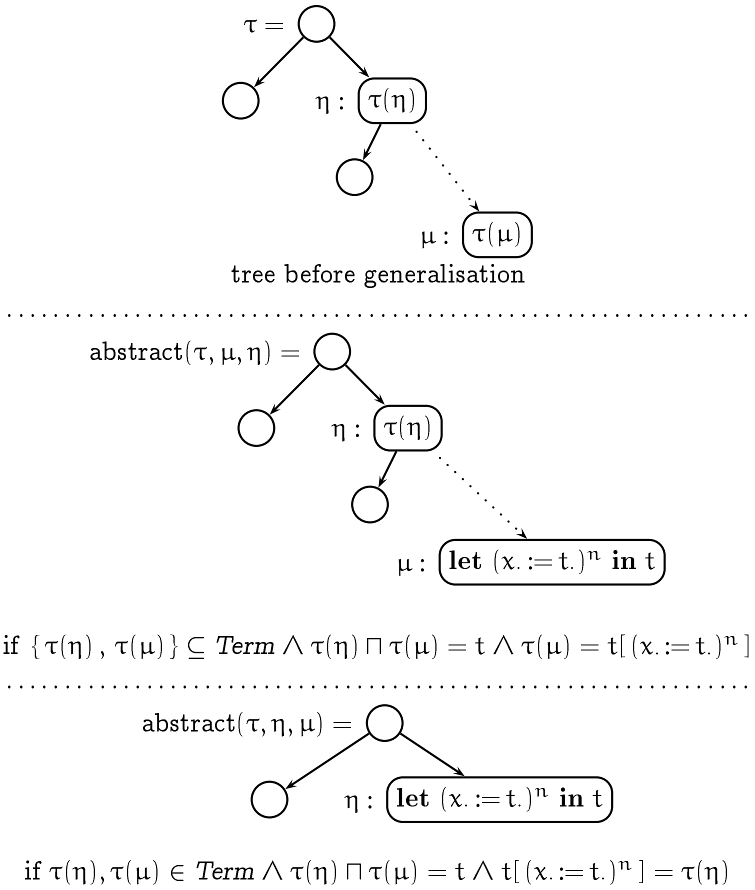


tree before operation

if $\{\, t^n \,\} = \{\, t \mid \tau(\mu) \xrightarrow[\mathsf{posscp}(q)]{} t \,\}$

if $\tau(\mu) = h\ t^n \wedge h \in \mathit{Constructor} \cup \mathit{Function} \cup \mathit{Matcher} \cup \{\, \textbf{ifthenelse} \,\}$

Figure 5.7: Driving and splitting in positive supercompilation.



tree before generalisation

if $\{\,\tau(\eta)\,,\,\tau(\mu)\,\} \subseteq \mathit{Term} \wedge \tau(\eta) \sqcap \tau(\mu) = t \wedge \tau(\mu) = t[\,(x. := t.)^n\,]$

if $\tau(\eta), \tau(\mu) \in \mathit{Term} \wedge \tau(\eta) \sqcap \tau(\mu) = t \wedge t[\,(x. := t.)^n\,] = \tau(\eta)$

Figure 5.8: Downwards and upwards generalisation in positive supercompilation.

rightmost node one step further is the very essence of removing the intermediate data structure! I will thus need a more sophisticated way of determining when to stop. □

To ensure termination and at the same time provide reasonable specialisation, I partition the nodes in the transformation trace tree into three categories. A node η is

1. a **let nodes**, if $\tau\eta$ is labelled by a **let**-term;
2. a **global nodes**, if $\tau\eta \xrightarrow[\mathsf{pscpout}(q)]{}_{\theta} t'$ such that $\theta \neq [\,]$ (for some t');
3. a **local nodes**, otherwise.

Global nodes are those nodes that represent speculative execution. Local nodes are those nodes that are not global and does not contain **let**-terms. This partitioning of the nodes is used to control the unfolding by only comparing *relevant* ancestors.

Definition 37 (Relevant ancestor) Let τ be a transformation trace tree and let the set of *relevant ancestors* $\mathrm{relanc}(\tau, \eta)$ of a node η in τ be defined thus:

$$\mathrm{relanc}(\tau,\eta) \overset{\mathrm{def}}{=} \begin{cases} \emptyset, & \text{if } \eta \text{ let} \\ \{\, \mu \mid \mu <_\tau \eta \wedge \mu \text{ is global} \,\}, & \text{if } \eta \text{ global} \\ \{\, \mu \mid \mu <_\tau \eta \wedge \forall \nu \; (\, \mu <_\tau \nu <_\tau \eta \Rightarrow \nu \text{ not global} \,) \,\}, & \text{if } \eta \text{ local} \end{cases}$$

□

That is, the *local ancestors* of a local node η are all ancestors up to the first global ancestor.

By using generalisation operations to split up terms in smaller parts while driving the program, I can use stop criteria very similar to the ones used in deforestation: A node needs no more processing when it consists entirely of constructors and variables, or when it is a renaming of an ancestor. When a node needs no more processing, I say that it is *finished*.

Definition 38 (Finished) A leaf μ in a transformation trace tree τ is *finished* if either $\tau\mu \in \mathit{Passive}$ or there exists an ancestor $\eta <_\tau \mu$ such that $\tau\eta \equiv \tau\mu$. A tree τ is said to be *finished* when all leaves are finished. □

```
                    $\tau \in$ Tree Term, $\{\eta\,,\ \mu\} \in \mathscr{D}\tau$

$M_+\ \tau =$ if $\tau$ finished then $\tau$
          else let $\mu$ suchthat $\mu \in \text{leaves}_\tau \wedge \tau\mu$ not finished
                    problems $:= \{\eta \mid \eta \in \text{relanc}(\tau, \mu) \wedge \tau\eta \trianglelefteq_{\text{hom}} \tau\mu\}$
             in if problems $= \varnothing$ then drive$(\tau, \mu)$
                else let $\eta$ suchthat $\eta \in$ problems
                     in if $\tau\mu \geq_{\text{inst}} \tau\eta$ then abstract$(\tau, \mu, \eta)$
                        else if $\tau\mu \| \tau\eta$ then split$(\tau, \mu)$
                             else abstract$(\tau, \eta, \mu)$
```

Figure 5.9: Positive supercompilation. Starting with a singleton tree labelled by the initial term, repeated application of M_+ will eventually produce a finished tree.

Definition 39 (Positive SCP) For $q \in$ *Program* and $t \in$ *Term*, the positive-supercompilation transformation-trace tree is obtained by repeated application of the function $M_+ \in$ *Tree Term* ⇨ *Tree Term* defined in Figure 5.9 to the singleton tree labelled by t. □

5.4 Further Reading and Comparison

Supercompilation is a shortening of *supervised compilation* and is the basis of *meta-computation*, an approach to artificial intelligence conceived by Valentin Turchin and co-workers. A good starting point for understanding the ideas behind meta-computation is his 1986 paper [115], where he tries to formulate supercompilation in more familiar terms than what had previously been published in the literature. For a formulation in a setting like the one presented in this chapter, see Glück and Sørensen [38]. Sørensen discovered in his Master's Thesis [103] that the homeomorphic embedding relation could be used to characterise and avoid non-termination.

For a treatment of unification, see Lassez, Maher and Marriott [61]. A linear-time unification algorithm was presented by Paterson and Wegman [79] based on

Tarjan's union–find algorithm [113]. A more simple, but yet effective, algorithm was proposed by Corbin and Bidoit [23].

Partial evaluation and supercompilation is quite different, although they aim at the same things. In partial evaluation, one usually transforms nested function calls inside-out, or one simply transforms the calls separately. Moreover, partial evaluation is traditionally based on constant propagation, not unification-based propagation like the one I have described in this chapter. Compared to deforestation, positive supercompilation essentially only differs in the *way* information is propagated. More specifically, in supercompilation the substitution derived by $\xrightarrow[\mathsf{pscpin}(q)]{}_{\theta}$ is applied to the whole term by the rule PROP. This additional information propagation gives extra transformation strength, as can be seen from the string-matcher example. In this respect, supercompilation is "stronger" than both deforestation and partial evaluation. A similarly strong transformation technique is *generalized partial computation*, conceived by Futamura and Nogi [32] and later extended by Takano [109, 110], in which axioms for algebraic data types and properties of primitive operators are taken into account during driving of a program. However, the main advantage of positive supercompilation, as presented in this chapter, compared to other techniques, is that the transformation is guaranteed to terminate (see Sørensen [108] and Chapter 9 for a proof).

Positive supercompilation has, of course, also disadvantages compared to other techniques. The main drawback is that the technique is inherently *online*, which seems to prohibit an effective offline binding-time annotation of programs, as seen in partial evaluation or type-based deforestation. Moreover, the homeomorphic-embedding strategy employed in positive supercompilation to avoid non-termination is rather expensive to compute, since each new leaf in the transformation trace tree potentially has to be compared to all its ancestors (each comparison takes time proportional to the product of the term sizes). In Chapter 10, I will remedy this last point by showing how to avoid most unnecessary comparisons

Another disadvantage of positive supercompilation, which it shares with most other program transformation techniques, is that there is no guarantee that a transformed program is at least as efficient as the original program. The main reason for this deficiency is that unfolding might duplicate terms representing computations. In Chapter 9, I will return to this issue.

Chapter 6

Perfect Supercompilation

In this chapter I will present an extension of positive supercompilation as seen in the previous chapter. The reason that it is called *perfect supercompilation* is that the underlying machinery for propagating information through the transformation trace is *perfect*, in contrast to that of the previous chapter where only positive information was used during transformation. I will present a rather general framework for the problem of handling negative (as well as positive) information about terms.

Example 21 (Negative information needed) Consider a program that tests for membership in a list:

$$\begin{array}{ll} \mathit{member}\ [\,]\ x & = \textsc{false} \\ \mathit{member}\ (y : \mathit{ys})\ x & = \textbf{if}\ x == y\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \mathit{member}\ \mathit{ys}\ x\ \ . \end{array}$$

If I want to transform this program with respect to the fixed list, e.g.,

$$\mathit{member}\ [\,\textsc{a}, \textsc{a}, \textsc{b}, \textsc{b}\,]\ x\ \ ,$$

by *positive* supercompilation, I will get the tree shown in Figure 6.1. From this tree I get the following program.

$$\begin{array}{ll} \mathit{main}\ x = h_1\ x & \\ h_1\ x & = \textbf{if}\ x == \textsc{a}\ \textbf{then}\ \textsc{true}\ \textbf{else}\ h_2\ x \\ h_2\ x & = \textbf{if}\ x == \textsc{a}\ \textbf{then}\ \textsc{true}\ \textbf{else}\ h_3\ x \\ h_3\ x & = \textbf{if}\ x == \textsc{b}\ \textbf{then}\ \textsc{true}\ \textbf{else}\ h_4\ x \\ h_4\ x & = \textbf{if}\ x == \textsc{b}\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \textsc{false}\ \ . \end{array}$$

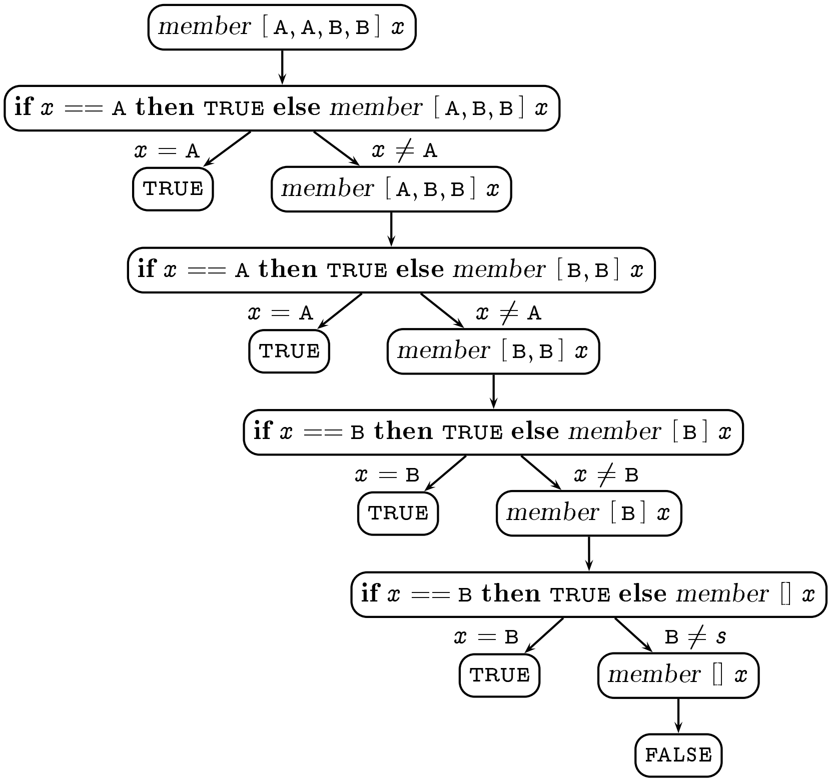

member [A, A, B, B] *x*
if *x* == A **then** TRUE **else** *member* [A, B, B] *x*
x = A
x ≠ A
TRUE
member [A, B, B] *x*
if *x* == A **then** TRUE **else** *member* [B, B] *x*
x = A
x ≠ A
TRUE
member [B, B] *x*
if *x* == B **then** TRUE **else** *member* [B] *x*
x = B
x ≠ B
TRUE
member [B] *x*
if *x* == B **then** TRUE **else** *member* [] *x*
x = B
B ≠ *s*
TRUE
member [] *x*
FALSE

Figure 6.1: Driving without negative information propagation results in redundant test.

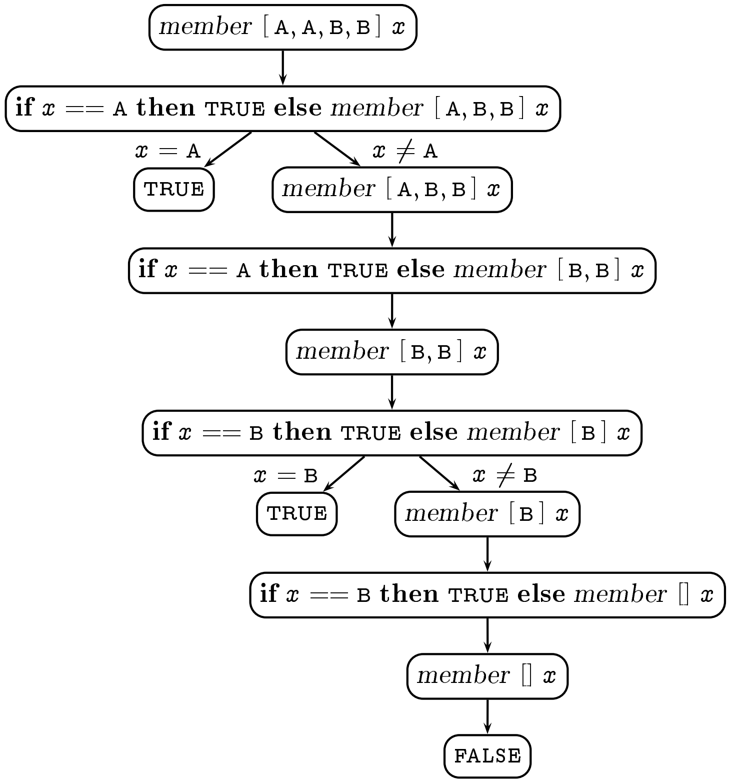

member [A, A, B, B] x
if x == A then TRUE else member [A, B, B] x
x = A
x ≠ A
TRUE
member [A, B, B] x
if x == A then TRUE else member [B, B] x
member [B, B] x
if x == B then TRUE else member [B] x
x = B
x ≠ B
TRUE
member [B] x
if x == B then TRUE else member [] x
member [] x
FALSE

Figure 6.2: Utilising negative information removes redundant tests.

The problem with this program is that functions h_2 and h_4 are superfluous. If negative information had been propagated and used to control the unfolding process, I would have gotten a transformation trace like the one in Figure 6.2, from which a program like following could be extracted:

$$\begin{aligned} &\mathit{main}\ x = h_1\ x \\ &h_1\ x \quad = \textbf{if}\ x == \text{A}\ \textbf{then}\ \text{TRUE}\ \textbf{else}\ h_3\ x \\ &h_3\ x \quad = \textbf{if}\ x == \text{B}\ \textbf{then}\ \text{TRUE}\ \textbf{else}\ \text{FALSE}\ \ . \end{aligned}$$

□

6.1 Negative Information Propagation

The preceding example illustrates that *negative* information is needed when driving a program. As in the previous chapter, consider the non-ground term

$$\textbf{if } x == y \textbf{ then } \mathrm{t} \textbf{ else } \mathrm{t}' \tag{6.1}$$

where t and t' are two unspecified terms. Again, the conditional cannot be directly executed, since I cannot decide whether or not x is equal to y. And, again, *if* x and y evaluates to the same value, I can safely assume that it must hold that any y can be replaced by x in the **then**-branch (i.e., in t). But conversely, *if* x and y are different, I can also safely assume that it must hold that any y is *distinct* from any x in the **else**-branch (i.e., in t'). I therefore want to transform the term in Equation 6.1 into

$$\textbf{if } x == y \textbf{ then } \mathrm{t}[\,y := x\,] \textbf{ else } \mathrm{t}'[\,x \neq y\,]\ .$$

The problem is that the *disequations*[1] $x \neq y$ cannot be propagated to the **else**-branch in form of a substitution. Moreover, the notions of unifiability and disunifiability (cf. Definition 28) do not suffice to control which branches should or should not be produced in the transformation trace.

I thus need a general device that can hold information about what choices have been made during driving. Each time I want to speculatively unfold a term, I want to add new information to this device, and I want to be able to ask whether such newly obtained knowledge contradicts the information I have previously collected.

Since comparisons in my language are performed on passive terms, it would be natural to represent such information as a conjunction of equations and disequations on passive terms:

$$\bigwedge_{i=1}^{n} a_i = a_i' \wedge \bigwedge_{i=1}^{m} a_i \neq a_i'\ .$$

I will say that such a conjunction is *weakly consistent* if there exists a substition θ such that, for every equation $a = a'$, $a\theta$ is syntactically equal to $a'\theta$ and, for every disequation $a \neq a'$, $a\theta$ is syntactically different from $a'\theta$. Intuitively, this

[1] It is called a disequation to distinguish it from inequalities like $x < y$.

makes sense because the existence of such a substitution implies the existence of a set of real computations where the uninstantiated variables have been bound to values.

The problem is now to find a way to calculate the consistency of a conjunction of equations and disequations. For this purpose, I will introduce a more rigorous system that both can hold the collected information and for which consistency can easily be checked. To avoid confusion between mathematical equality and constraint involving syntactic equality, I will use $\doteq$ and $\dot{\neq}$ for the latter.

6.2 Restriction Systems

Definition 40 A *restriction system* $\mathsf{R} \in \mathit{Restriction}$ is defined by the grammar

$$
\begin{array}{lclcll}
\mathit{Restriction} & \ni & \mathsf{R} & ::= & \mathsf{a} \doteq \mathsf{a}' & \text{(equation)} \\
 & & & | & \mathsf{R}' \wedge \mathsf{R}'' & \text{(conjunction)} \\
 & & & | & \mathsf{D} & \\
 & & \mathsf{D} & ::= & \bot & \text{(false)} \\
 & & & | & \top & \text{(true)} \\
 & & & | & \mathsf{a} \mathrel{\dot{\neq}} \mathsf{a}' & \text{(disequation)} \\
 & & & | & \mathsf{D}' \vee \mathsf{D}'' & \text{(disjunction)} \\
\mathit{Passive} & \ni & \mathsf{a}, \mathsf{b} & ::= & \mathsf{x} & \text{(variable)} \\
 & & & | & \mathsf{c}\ \mathsf{a}^{\mathsf{n}} & \text{(constructor)}
\end{array}
$$

where $\mathsf{x} \in \mathit{Variable}$ and $\mathsf{c} \in \mathit{Constructor}$. □

A restriction system only treats passive terms, or *constructor terms* as it were, because the driving algorithm only need to propagate such information, as you have seen in the previous chapter. Observe that the conjunctions of equations and disequations that was sugested as a natural representation of collected information can be represented in a restriction system. The extra stuff, namely disjunctions, $\bot$ and $\top$, will be used when I want to test the consistency of a restriction system. I will now make precise what I mean by weak consistency of a restriction system.

Definition 41 (Weak Consistency) Let *Substitution* be the set of all idempotent substitutions that map variables to passive terms. A *weak solution* to restriction system $\mathsf{R} \in \mathit{Restriction}$, denoted $\mathsf{weaksol}(\mathsf{R})$, is a subset of *Substitution* defined in the following way.

1. $\mathsf{weaksol}(\top) \stackrel{\text{def}}{=} \mathit{Substitution}$
2. $\mathsf{weaksol}(\bot) \stackrel{\text{def}}{=} \varnothing$
3. $\mathsf{weaksol}(a \doteq a') \stackrel{\text{def}}{=} \{\, \theta \in \mathit{Substitution} \mid a\theta = a'\theta \,\}$
4. $\mathsf{weaksol}(a \neq a') \stackrel{\text{def}}{=} \{\, \theta \in \mathit{Substitution} \mid a\theta \neq a'\theta \,\}$
5. $\mathsf{weaksol}(\mathsf{R} \wedge \mathsf{R}') \stackrel{\text{def}}{=} \mathsf{weaksol}(\mathsf{R}) \cap \mathsf{weaksol}(\mathsf{R}')$
6. $\mathsf{weaksol}(\mathsf{D} \vee \mathsf{D}') \stackrel{\text{def}}{=} \mathsf{weaksol}(\mathsf{D}) \cup \mathsf{weaksol}(\mathsf{D}')$

I say that R is *weakly consistent* iff $\mathsf{weaksol}(\mathsf{R}) \neq \varnothing$. □

To find out whether a restriction system is weakly consistent, I will proceed in two steps. First I will bring the system to a *weak normal form* (to be defined below) by a number of rewrite steps. When the restriction system is on weak normal form, a simple syntactic check can be used to test for consistency.

Definition 42 (Weak normal form)

1. A restriction system $\mathsf{R} \in \mathit{Restriction}$ is on *quasi-weak normal form* iff it is either $\top$, $\bot$ or of the form

   $$\left(\bigwedge_{i=1}^{n} x_i \doteq a_i\right) \wedge \bigwedge_{j=1}^{m} \left(\bigvee_{k=1}^{l} y_{j,k} \neq a_{j,k}\right) \quad ,$$

   where $\{\, x \,,\, y \,\} \subseteq \mathit{Variable}$. Moreover,

   (a) A variable x_i in R is a *solved variable* iff it occurs exactly once in R.

   (b) A variable $y_{j,k}$ in R is a *semi-solved variable* iff occurs exactly once in $\left(\bigvee_{k=1}^{l} y_{j,k} \neq a_{j,k}\right)$.

2. R is in *weak normal form* iff R is in quasi-weak normal form and every x_i is a solved variable and every $y_{j,k}$ is a semi-solved variable. □

Example 22 (Weak nomal form) Consider the restriction system

$$\textsc{pair}\ (\textsc{cons}\ y\ z_1)\ z_1 \doteq \textsc{pair}\ x\ z_2 \wedge \textsc{pair}\ y\ z_3 \mathrel{\dot{\neq}} \textsc{pair}\ (\textsc{cons}\ z_3\ z_1)\ z_2 \quad .$$

A weak normal form for this system is

$$x \doteq \textsc{cons}\ y\ z_1 \wedge z_2 \doteq z_1 \wedge (y \mathrel{\dot{\neq}} \textsc{cons}\ z_3\ z_3\ \vee\ z_1 \mathrel{\dot{\neq}} z_3) \quad ,$$

where x and z_2 are solved, and y and z_1 are semi-solved. Another weak normal form is

$$x \doteq \textsc{cons}\ y\ z_2 \wedge z_1 \doteq z_2 \wedge (y \mathrel{\dot{\neq}} \textsc{cons}\ z_2\ z_2\ \vee\ z_3 \mathrel{\dot{\neq}} z_2) \quad ,$$

where x and z_1 are solved, and y and z_3 are semi-solved. Hence there is in general no unique weak normal form for a restriction system. □

The reason for making all the above definitions is that when a restriction is in weak normal form, it is very easy to decide whether it is weakly consistent.

Proposition 4 (Consistency check) *A restriction system* R *in weak normal form is weakly consistent iff* $R \neq \bot$.

PROOF (sketch).

($\Rightarrow$) If R is weakly consistent, there exists a substitution θ such that $\theta \in \mathrm{weaksol}(R)$. Thus R cannot be $\bot$ since $\mathrm{weaksol}(\bot) = \varnothing$.

($\Leftarrow$) If R is $\top$, then $\mathrm{weaksol}(R) = \mathit{Substitution} \neq \varnothing$. Otherwise, assume $R = \bigwedge_{i=1}^{n} x_i \doteq a_i \wedge \bigwedge_{j=1}^{m} D_j$. Then the substitution $[\,(x_{.} := a_{.})^n\,]$ will be contained in weaksol(R) since each x_i occurs only once and for every disequation $y \neq b$ I have that $y \notin \mathscr{V}b$ which trivially makes $y \neq b$. □

The rewrite rule defined in Figure 6.3 can be used to turn a restriction system into a normal form. Not surprisingly, these rewrite rules are a superset of those one could use to find a most general unifier.

Lemma 13 (Normalisation) *Exhaustive application of the rewrite rule defined in Figure 6.3 to any restriction system* R *will result in a restriction system in weak normal form.*

$$
\begin{array}{llcll}
(\mathrm{T1}) & x \doteq x & \xrightarrow[\mathsf{rsys}]{} & \top & \\
(\mathrm{T2}) & x \not\doteq x & \xrightarrow[\mathsf{rsys}]{} & \bot & \\
(\mathrm{B1}) & R \wedge \top & \xrightarrow[\mathsf{rsys}]{} & R & \\
(\mathrm{B2}) & D \vee \bot & \xrightarrow[\mathsf{rsys}]{} & D & \\
(\mathrm{B3}) & R \wedge \bot & \xrightarrow[\mathsf{rsys}]{} & \bot & \\
(\mathrm{B4}) & D \vee \top & \xrightarrow[\mathsf{rsys}]{} & \top & \\
(\mathrm{C1}) & c\ b^n \doteq c'\ a^m & \xrightarrow[\mathsf{rsys}]{} & \bot & (\text{if } c \neq c') \\
(\mathrm{C2}) & c\ b^n \not\doteq c'\ a^m & \xrightarrow[\mathsf{rsys}]{} & \top & (\text{if } c \neq c') \\
(\mathrm{D1}) & c\ b^n \doteq c\ a^n & \xrightarrow[\mathsf{rsys}]{} & b_1 \doteq a_1 \wedge \cdots \wedge b_n \doteq a_n & \\
(\mathrm{D2}) & c\ b^n \not\doteq c\ a^n & \xrightarrow[\mathsf{rsys}]{} & b_1 \not\doteq a_1 \vee \cdots \vee b_n \not\doteq a_n & \\
(\mathrm{O1}) & x \doteq a & \xrightarrow[\mathsf{rsys}]{} & \bot & (\text{if } x \in \mathscr{V}a \,\&\, a \notin \textit{Variable}) \\
(\mathrm{O2}) & x \not\doteq a & \xrightarrow[\mathsf{rsys}]{} & \top & (\text{if } x \in \mathscr{V}a \,\&\, a \notin \textit{Variable}) \\
(\mathrm{R1}) & x \doteq a \wedge R & \xrightarrow[\mathsf{rsys}]{} & x \doteq a \wedge R[x := a] & \\
 & & & \multicolumn{2}{r}{(\text{if } x \in \mathscr{V}R \,\&\, x \notin \mathscr{V}a \,\&\, (a \in \textit{Variable} \Rightarrow a \in \mathscr{V}R))} \\
(\mathrm{R2}) & x \not\doteq a \vee D & \xrightarrow[\mathsf{rsys}]{} & x \not\doteq a \vee D[x := a] & \\
 & & & \multicolumn{2}{r}{(\text{if } x \in \mathscr{V}D \,\&\, x \notin \mathscr{V}a \,\&\, (a \in \textit{Variable} \Rightarrow a \in \mathscr{V}D))}
\end{array}
$$

Figure 6.3: Normalisation of restriction systems. A guard to the right of rule has to be satisfied to enable the rule.

PROOF (sketch). First show that exhaustive application of the rewrite rules to any restriction system terminates. This is done by defining three functions.

1. $\phi_1 \in \mathit{Restriction} \to \mathbb{N}$ is the number of variables in R that are not solved.

2. $\phi_2 \in \mathit{Restriction} \to \mathbb{N}$ is the number of variables in R that are not semi-solved.

3. $\phi_3 \in \mathit{Restriction} \to \mathbb{N}$ is the size of R.

With these three functions, define a function $\Phi(R) \overset{\text{def}}{=} \langle \phi_1(R)\,,\ \phi_2(R)\,,\ \phi_3(R)\rangle$ and show that this ordered tuple decreases by application of any rewrite rule. Since such a decrease cannot go on forever, rewriting will terminate.

Then consider a restriction system that is not in weak normal form and show that in such a case there will always be some rewrite that applies. □

Proposition 5 (Solution preserving) *The rewrite rules defined in Figure 6.3 are* solution preserving, *that is, if* $R \xrightarrow[\text{rsys}]{} R'$*, then* $\theta \in \mathit{weaksol}(R)$ *iff* $\theta \in \mathit{weaksol}(R')$.

PROOF (sketch). Most of the rules only rewrite a local part of the system and thus the context is unchanged. Rules T1–O2 are straight forward. For R1, let $R = (x \doteq a \wedge \bigwedge_{i=1}^{n} R_i) \xrightarrow[\text{rsys}]{} (x \doteq a \wedge \bigwedge_{i=1}^{n} R_i[\,x := a\,]) = R'$ and suppose $\theta \in \text{weaksol}(R)$. Then $x\theta = a\theta$. For each R_i of the form $a_i \doteq a_i'$, it will be the case that

$$\begin{aligned} R_i\theta &= a_i\theta \doteq a_i'\theta \\ &= (a_i\theta)[\,x := a\,] \doteq (a_i'\theta)[\,x := a\,] \\ &\quad \text{(since } x[\,x := a\,] = a[\,x := a\,] \text{ and } \theta \text{ is idempotent)} \\ &= (R_i\theta)[\,x := a\,]\ , \end{aligned}$$

and so forth for the other forms.

For R2, let $R = (x \not\doteq a \vee \bigvee_{i=1}^{n} R_i) \xrightarrow[\text{rsys}]{} (x \not\doteq a \vee \bigvee_{i=1}^{n} R_i[\,x := a\,]) = R'$. Then any $\theta \in \text{weaksol}(R)$ will either make $x\theta \neq a\theta$ or $x\theta = a\theta$. In the former case I trivially have that $\theta \in \text{weaksol}(R')$. In the latter case, proceed as with R1. □

The solutions to a restriction system that I have considered so far are called weak because, even though a restriction system is different from $\bot$, there might not exist a type-correct *value* substitution which satisfies the constraints defined by the restriction system. In other words, an algebraic data type might naturally restrict the number of different values that variables of that type can hold. Consider, for instance, three boolean variables x, y, and z for which the weak-normal-form restriction system

$$x \neq y \wedge y \neq z \wedge z \neq x$$

has to hold. There is no type-correct substitution of values that will satisfy this system. In general, I would have to try out all possible combinations of value assignments for such variables of *finite-value* types, and subsequently perform normalisation of the restriction systems I obtain from such assignments. I will call a restriction system *satisfiable* if all such subsequent normalisations do not result in $\perp$. Observe that strong consistency is NPTIME-hard.

In summary, in this section I have described a machinery that, given information about the choices made so far, can tell us whether or not a particular branch in the transformation trace is worth taking. I will now, similarly to what was seen in the previous chapters, instrument the semantics of our language so that it can handle non-ground terms, but now I will use restriction systems to control the unfolding. Again, this extension will enable me to produce transformation trace trees from which I can extract new programs.

6.3 Driving with Restriction Systems

I will now attach a restriction system to every term in the transformation trace tree. The initial term (the root of the tree) will have the empty restriction system $\top$ attached to it to indicate that no information has been gathered yet. As driving proceeds, more information will be accumulated and therefore the restriction system associated with each new term in the tree will grow.

Definition 43 (Perfect driving) For a program q, I define the relation $\xrightarrow[\mathsf{perscp}(q)]{}$ on $(\mathit{Term}^{\mathrm{let}} \times \mathit{Restriction}) \times (\mathit{Term}^{\mathrm{let}} \times \mathit{Restriction})$ by the inference system in Figures 6.4 and 6.5. □

The consistency check in rules C and E–F are done as described in the previous section, that is, the restriction system is normalised and it is tested whether it becomes $\perp$. Rule K extracts the positive information from the restriction system (by normalising it) and applies it (as a substitition) to the

$$(\mathrm{A})\quad \dfrac{f\ x^n \overset{q}{=} t}{\langle f\ t^n\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle t[\,(x_.:=t_.)^n\,]\,,\ R\rangle}$$

$$(\mathrm{B})\quad \dfrac{g\,(c\ x^m)\ x_{m+1}\ldots x_n \overset{q}{=} t}{\langle g\,(c\ t^m)\ t_{m+1}\ldots t_n\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle t[\,(x_.:=t_.)^n\,]\,,\ R\rangle}$$

$$(\mathrm{C})\quad \dfrac{g\ p\ x^n \overset{q}{=} t \qquad R' = R \wedge [x \doteq p] \qquad R' \text{ satisfiable}}{\langle g\ x\ t^n\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle t[\,(x_.:=t_.)^n\,]\,,\ R'\rangle}$$

$$(\mathrm{D})\quad \dfrac{g\ p\ x^n \overset{q}{=} t \qquad \langle t\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle t'\,,\ R'\rangle}{\langle g\ t\ t^n\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle g\ t'\ t^n\,,\ R'\rangle}$$

$$(\mathrm{E})\quad \dfrac{R' = R \wedge [a \doteq a'] \qquad R' \text{ satisfiable}}{\langle \mathbf{if}\ a == a'\ \mathbf{then}\ t\ \mathbf{else}\ t'\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle t\,,\ R'\rangle}$$

$$(\mathrm{F})\quad \dfrac{R' = R \wedge [a \not\doteq a'] \qquad R' \text{ satisfiable}}{\langle \mathbf{if}\ a == a'\ \mathbf{then}\ t\ \mathbf{else}\ t'\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle t'\,,\ R'\rangle}$$

$$(\mathrm{G})\quad \dfrac{\langle t_1\,,\ R\rangle \xrightarrow[\mathsf{perout}(q)]{} \langle t_1'\,,\ R'\rangle}{\langle \mathbf{if}\ t_1 == t_2\ \mathbf{then}\ t_3\ \mathbf{else}\ t_4\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle \mathbf{if}\ t_1' == t_2\ \mathbf{then}\ t_3\ \mathbf{else}\ t_4\,,\ R'\rangle}$$

$$(\mathrm{H})\quad \dfrac{\langle t_2\,,\ R\rangle \xrightarrow[\mathsf{perout}(q)]{} \langle t_2'\,,\ R'\rangle}{\langle \mathbf{if}\ a == t_2\ \mathbf{then}\ t_3\ \mathbf{else}\ t_4\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle \mathbf{if}\ a == t_2'\ \mathbf{then}\ t_3\ \mathbf{else}\ t_4\,,\ R'\rangle}$$

$$(\mathrm{I})\quad \dfrac{\langle t\,,\ R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle t'\,,\ R'\rangle}{\langle t\,,\ R\rangle \xrightarrow[\mathsf{perout}(q)]{} \langle t'\,,\ R'\rangle}$$

$$(\mathrm{J})\quad \dfrac{\langle t\,,\ R\rangle \xrightarrow[\mathsf{perout}(q)]{} \langle t'\,,\ R'\rangle}{\langle c\ a^{m-1}\ t\ t_{m+1}\ldots t_n\,,\ R\rangle \xrightarrow[\mathsf{perout}(q)]{} \langle c\ a^{m-1}\ t'\ t_{m+1}\ldots t_n\,,\ R'\rangle}$$

Figure 6.4: Inner unfold rules for perfect driving using restriction systems.

$$(\mathrm{K})\ \frac{\langle t\,,\,R\rangle \xrightarrow[\mathsf{perin}(q)]{} \langle t'\,,\,R'\rangle \qquad R' \xrightarrow[\mathsf{rsys}]{*} (\bigwedge_{i=1}^{n} x_i \doteq a_i \wedge \bigwedge_{j=1}^{m} D_j)}{\langle t\,,\,R\rangle \xrightarrow[\mathsf{perscp}(q)]{} \left\langle t'[\,(x.\!:=a.)^n\,]\,,\,\bigwedge_{j=1}^{m} D_j\right\rangle}$$

$$(\mathrm{L})\ \frac{\langle t\,,\,R\rangle \xrightarrow[\mathsf{perscp}(q)]{} \langle t'\,,\,R'\rangle}{\left\langle c\ a^{m-1}\ t\ t_{m+1}\ldots t_n\,,\,R\right\rangle \xrightarrow[\mathsf{perscp}(q)]{} \left\langle c\ a^{m-1}\ t'\ t_{m+1}\ldots t_n\,,\,R'\right\rangle}$$

$$(\mathrm{M})\ \frac{}{\mathbf{let}\ (x.:=t.)^{n-1}\ \mathbf{in}\ t_n \xrightarrow[\mathsf{perscp}(q)]{} t_i}\ i\in\{1,\ldots,n\}$$

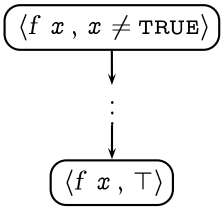
Figure 6.5: Outer unfold rules for perfect driving using restriction systems.

whole term. Observe that if no negative information was collected, I would achieve exactly the same result as with positive supercompilation.

Example 23 Consider again the member-test program from Example 21. By using the newly defined notion of perfect driving, the transformation trace of the term '*member* [A, A, B, B] x' would be like the one in Figure 6.2 (if I omit showing the restriction systems). □

By introducing restriction systems, I have made driving more powerful. But there is no such thing as a free lunch: I have introduced a new source of nontermination. The problem is that it is no longer sufficient simply to compare terms when deciding when to fold. The reason is that a node in the transformation trace tree no longer is fully described by the term it contains; the restriction system has to be taken into account.

Example 24 (Less restrictive) Consider a transformation trace tree



Since the term in the leaf is the same as the one in its ancestor, I would like to stop driving and make a fold operation such that the transformed program will make a recursive call. However, this is not safe. The ancestor assumes that variable x cannot have the value TRUE, but there is no such assumption made by the leaf. If I stopped driving the leaf, and generated a recursive call, the transformed program could end up in a configuration where it would get stuck or simply generate a wrong result if x did indeed have the value TRUE. □

The problem is that the ancestor is *more restrictive* than the leaf. I will therefore have to take the restriction systems into account when I decide when to stop and how to generalise. In this chapter I will choose a very simple solution: When comparing two nodes, I will only accept folding when the ancestor contains the empty restriction system $\top$, since $\top$ is more general than any other restriction system. If the ancestor contains a restriction system which is different from $\top$, I will replace this restriction system with $\top$ and restart driving at that node.

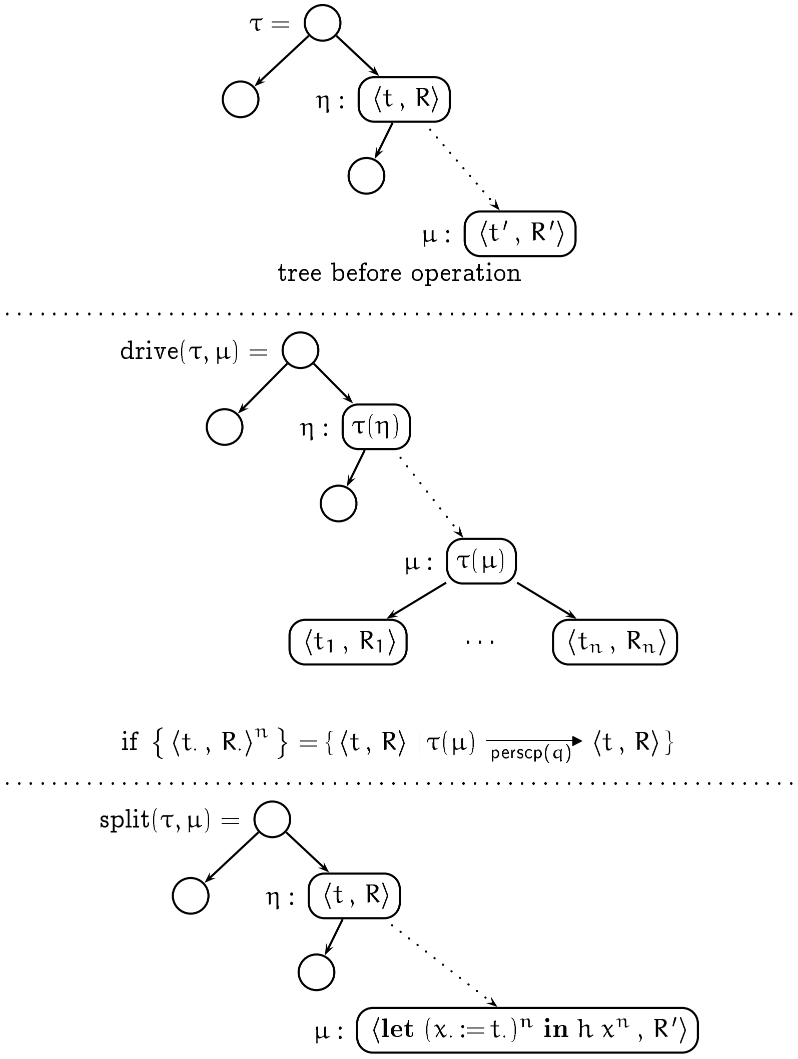
6.4 Fitting it together

With the above notion of generalisation on restriction systems, I can formulate the set of generalisation steps depicted in Figures 6.6 and 6.7. The new drop operation takes care of the previously mentioned situation.

The nodes in the transformation trace tree can now be partitioned similarly to what you have seen in Chapter 5 (cf. Definition 37), and I only need to modify the definition of when a node needs no more processing.

Definition 44 (Finished) Let τ be a transformation trace tree, $\mu \in \mathsf{leaves}_\tau$, and $\tau(\mu) = \langle \mathsf{t}\,,\, \mathsf{R}\rangle$. Leaf μ is *finished* iff $\mathsf{t} \in \mathit{Passive}$ or there exists an ancestor $\eta <_\tau \mu$ with $\tau(\eta) = \langle \mathsf{t}'\,,\, \mathsf{R}'\rangle$ such that $\mathsf{t}' \equiv \mathsf{t}$ and R is more restrictive than R'. A tree τ is said to be *finished* when all leaves are finished. □

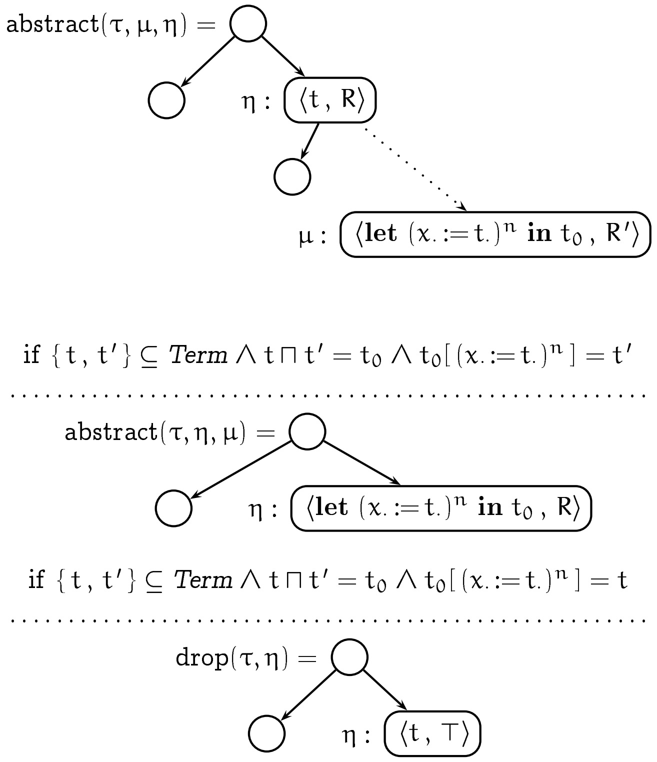
Definition 45 (Perfect supercompilation) For $\mathsf{q} \in \mathit{Program}$ and $\mathsf{t} \in \mathit{Term}$, the positive-supercompilation transformation-trace tree is obtained by repeated application of the function $\mathsf{M}_\pm \in \mathit{Tree}\,(\mathit{Term} \times \mathit{Restriction}) \Rightarrow \mathit{Tree}\,(\mathit{Term} \times \mathit{Restriction})$ defined in Figure 6.8 to the singleton tree labelled by $\langle \mathsf{t}\,,\, \top\rangle$. □


$\tau =$
$\eta : \langle t \,,\, R \rangle$
$\mu : \langle t' \,,\, R' \rangle$
tree before operation
$\mathrm{drive}(\tau, \mu) =$
$\eta : \tau(\eta)$
$\mu : \tau(\mu)$
$\langle t_1 \,,\, R_1 \rangle$
$\cdots$
$\langle t_n \,,\, R_n \rangle$
$\mathrm{split}(\tau, \mu) =$
$\eta : \langle t \,,\, R \rangle$
$\mu : \langle \mathbf{let}\ (x_. := t_.)^n\ \mathbf{in}\ h\ x^n \,,\, R' \rangle$

if $\left\{ \langle t_. \,,\, R_. \rangle^n \right\} = \{ \langle t \,,\, R \rangle \mid \tau(\mu) \xrightarrow[\mathsf{perscp}(q)]{} \langle t \,,\, R \rangle \}$

If $t' = h\ t^n \wedge h \in \mathit{Constructor} \cup \mathit{Function} \cup \mathit{Matcher} \cup \{ \mathbf{ifthenelse} \}$

Figure 6.6: Driving and splitting in perfect supercompilation.


abstract(τ, μ, η) =
η : ⟨t , R⟩
μ : ⟨**let** (x. := t.)ⁿ **in** t₀ , R′⟩
abstract(τ, η, μ) =
η : ⟨**let** (x. := t.)ⁿ **in** t₀ , R⟩
drop(τ, η) =
η : ⟨t , ⊤⟩

if $\{ t , t' \} \subseteq \mathit{Term} \wedge t \sqcap t' = t_0 \wedge t_0[\,(x_. := t_.)^n\,] = t'$

if $\{ t , t' \} \subseteq \mathit{Term} \wedge t \sqcap t' = t_0 \wedge t_0[\,(x_. := t_.)^n\,] = t$

Figure 6.7: Generalisation in perfect supercompilation.

```
$M_\pm\ \tau =$ if finished($\tau$) then $\tau$
        else let $\mu$ suchthat $\eta \in \text{leaves}_\tau \wedge \neg\text{finished}(\tau\mu)$
                $\tau\mu := \langle t\,,\, R\rangle$
                $\textit{problems} := \{\eta \mid \eta \in \text{relanc}(\tau,\mu) \wedge \tau\eta = u \wedge u \trianglelefteq_{\text{hom}} t\}$
            in if $\textit{problems} = \varnothing$ then drive($\tau,\mu$)
               else let $\langle t'\,,\, R'\rangle$ suchthat $\eta \in \textit{problems} \wedge \tau\eta = \langle t'\,,\, R'\rangle$
                    in if $t \equiv t'$ then drop($\tau,\eta$)
                       else if $t \geq_{\text{inst}} t'$ then abstract($\tau,\mu,\eta$)
                            else if $t \| t'$ then split($\tau,\mu$)
                                 else abstract($\tau,\eta,\mu$)
```

Figure 6.8: Perfect supercompilation.

Example 25 In the previous chapter you saw that positive supercompilation could transform a naïvely specialised string matcher into a finite state machine, but the transformed program was not optimal in the sense that superfluous tests were still present. Applying the perfect supercompilation algorithm to the matcher used on the specific pattern [A, A, B], I will get the transformation trace seen in Figure 6.10 and Figure 6.11. The program extracted from this tree is shown in Figure 6.9, with some unnecessary functions removed. □

6.5 Further Reading

The use of negative information in supercompilation was first described by Turchin[118, 116] for the language Refal, which has ordered pattern matches similar to what is seen in more well-known functional languages. When Turchin drives the definition that *follows* a pattern $x \to a$, the fact that x cannot possibly match the pattern a is recorded by means of a *restriction* $\#x \to a$, which is subsequently used to prune branches driving tree. Glück and Klimov [36] later defined the notion of *perfect driving*, where simple restrictions of the form $x \neq y$ were used. The present formulation of supercompilation with restriction systems and generalisation is due to Secher [94].

$$\begin{aligned}
&\mathit{main}\ s &&= m_{\mathsf{aab}}\ s\\
&m_{\mathsf{aab}}\ [\,] &&= \texttt{FALSE}\\
&m_{\mathsf{aab}}\ (s : ss) &&= h_{\mathsf{aab}}\ s\ ss\\
&h_{\mathsf{aab}}\ s\ ss &&= \mathbf{if}\ \texttt{A} == s\ \mathbf{then}\ m_{\mathsf{ab}}\ ss\ \mathbf{else}\ m_{\mathsf{aab}}\ ss\\
&m_{\mathsf{ab}}\ [\,] &&= \texttt{FALSE}\\
&m_{\mathsf{ab}}\ (s : ss) &&= h_{\mathsf{ab}}\ s\ ss\\
&h_{\mathsf{ab}}\ s\ ss &&= \mathbf{if}\ \texttt{A} == s\ \mathbf{then}\ m_{\mathsf{b}}\ ss\ \mathbf{else}\ m_{\mathsf{aab}}\ ss\\
&m_{\mathsf{b}}\ [\,] &&= \texttt{FALSE}\\
&m_{\mathsf{b}}\ (s : ss) &&= h_{\mathsf{b}}\ s\ ss\\
&h_{\mathsf{b}}\ s\ ss &&= \mathbf{if}\ \texttt{B} == s\ \mathbf{then}\ \texttt{TRUE}\ \mathbf{else}\ h_{\mathsf{ab}}\ s\ ss
\end{aligned}$$

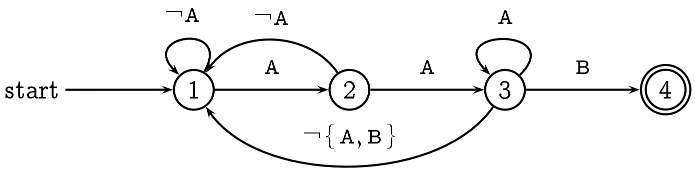


Figure 6.9: Optimal matcher obtained from specialising a general.

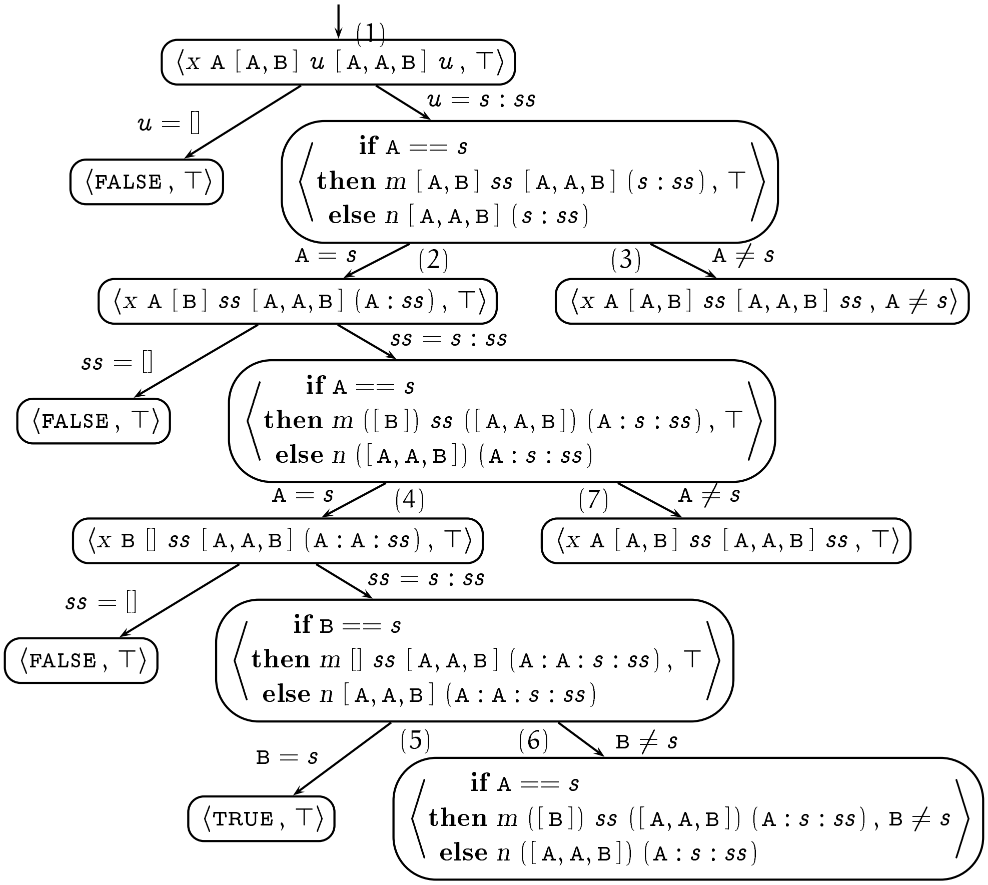

(1)
⟨x A [A, B] u [A, A, B] u , ⊤⟩
u = []
u = s : ss
⟨FALSE , ⊤⟩
⟨ if A == s then m [A, B] ss [A, A, B] (s : ss) , ⊤ else n [A, A, B] (s : ss) ⟩
A = s
(2)
(3)
A ≠ s
⟨x A [B] ss [A, A, B] (A : ss) , ⊤⟩
⟨x A [A, B] ss [A, A, B] ss , A ≠ s⟩
ss = []
ss = s : ss
⟨FALSE , ⊤⟩
⟨ if A == s then m ([B]) ss ([A, A, B]) (A : s : ss) , ⊤ else n ([A, A, B]) (A : s : ss) ⟩
A = s
(4)
(7)
A ≠ s
⟨x B [] ss [A, A, B] (A : A : ss) , ⊤⟩
⟨x A [A, B] ss [A, A, B] ss , ⊤⟩
ss = []
ss = s : ss
⟨FALSE , ⊤⟩
⟨ if B == s then m [] ss [A, A, B] (A : A : s : ss) , ⊤ else n [A, A, B] (A : A : s : ss) ⟩
(5)
(6)
B = s
B ≠ s
⟨TRUE , ⊤⟩
⟨ if A == s then m ([B]) ss ([A, A, B]) (A : s : ss) , B ≠ s else n ([A, A, B]) (A : s : ss) ⟩

Figure 6.10: Driving the string matcher with restriction systems. The local trees are shown in Figures 6.11, 6.12 and 6.13.

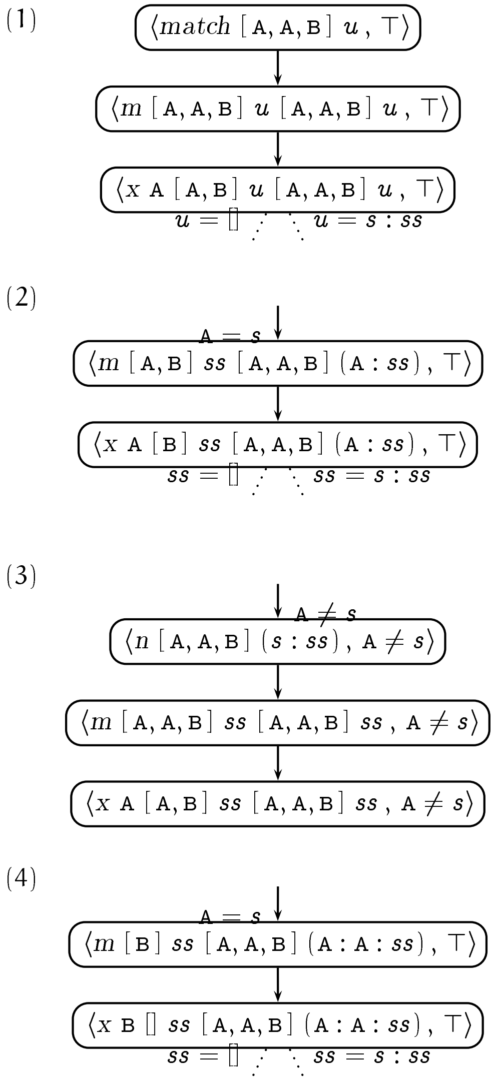

(1)
⟨match [A, A, B] u , ⊤⟩
⟨m [A, A, B] u [A, A, B] u , ⊤⟩
⟨x A [A, B] u [A, A, B] u , ⊤⟩
u = []
u = s : ss
(2)
A = s
⟨m [A, B] ss [A, A, B] (A : ss) , ⊤⟩
⟨x A [B] ss [A, A, B] (A : ss) , ⊤⟩
ss = []
ss = s : ss
(3)
A ≠ s
⟨n [A, A, B] (s : ss) , A ≠ s⟩
⟨m [A, A, B] ss [A, A, B] ss , A ≠ s⟩
⟨x A [A, B] ss [A, A, B] ss , A ≠ s⟩
(4)
A = s
⟨m [B] ss [A, A, B] (A : A : ss) , ⊤⟩
⟨x B [] ss [A, A, B] (A : A : ss) , ⊤⟩
ss = []
ss = s : ss

Figure 6.11: Local nodes in for the tree in Figure 6.10

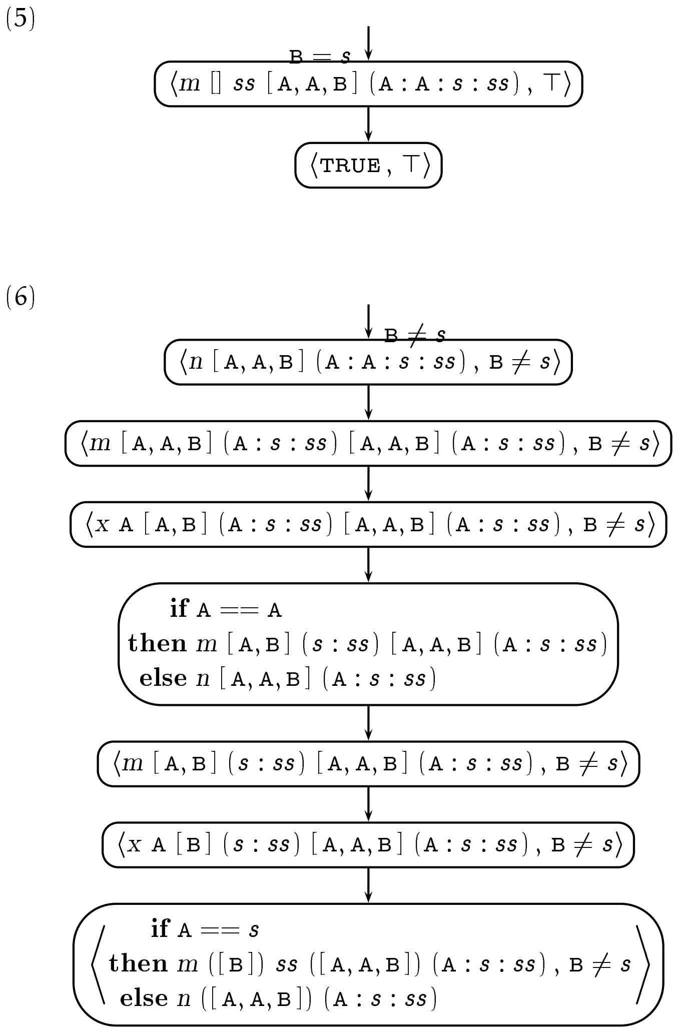

(5)
B = s
⟨m [] ss [A, A, B] (A : A : s : ss) , ⊤⟩
⟨TRUE , ⊤⟩
(6)
B ≠ s
⟨n [A, A, B] (A : A : s : ss) , B ≠ s⟩
⟨m [A, A, B] (A : s : ss) [A, A, B] (A : s : ss) , B ≠ s⟩
⟨x A [A, B] (A : s : ss) [A, A, B] (A : s : ss) , B ≠ s⟩
if A == A
then m [A, B] (s : ss) [A, A, B] (A : s : ss)
else n [A, A, B] (A : s : ss)
⟨m [A, B] (s : ss) [A, A, B] (A : s : ss) , B ≠ s⟩
⟨x A [B] (s : ss) [A, A, B] (A : s : ss) , B ≠ s⟩
if A == s
then m ([B]) ss ([A, A, B]) (A : s : ss) , B ≠ s
else n ([A, A, B]) (A : s : ss)

Figure 6.12: Local nodes in for the tree in Figure 6.10 (continued)

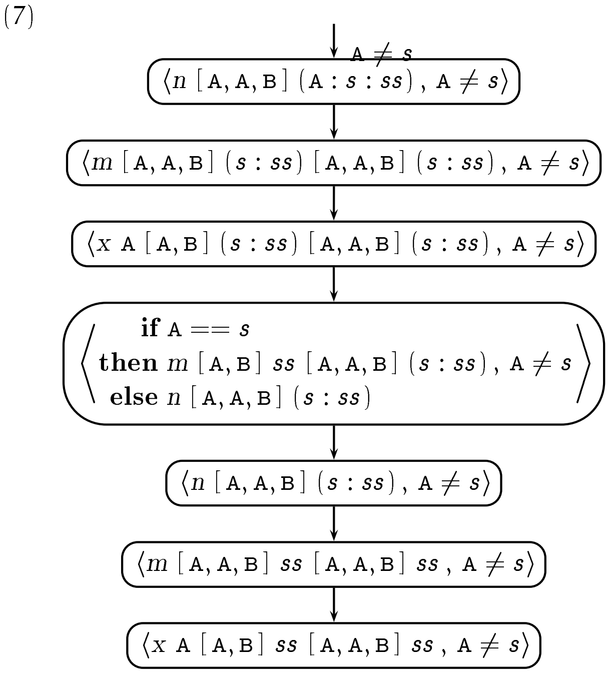

(7)
A ≠ s
⟨n [A, A, B] (A : s : ss) , A ≠ s⟩
⟨m [A, A, B] (s : ss) [A, A, B] (s : ss) , A ≠ s⟩
⟨x A [A, B] (s : ss) [A, A, B] (s : ss) , A ≠ s⟩
if A == s
then m [A, B] ss [A, A, B] (s : ss) , A ≠ s
else n [A, A, B] (s : ss)
⟨n [A, A, B] (s : ss) , A ≠ s⟩
⟨m [A, A, B] ss [A, A, B] ss , A ≠ s⟩
⟨x A [A, B] ss [A, A, B] ss , A ≠ s⟩

Figure 6.13: Local nodes in for the tree in Figure 6.10 (continued)

Equational theories on terms have been described by Comon, Kirchner and Lescanne [53, 20], and the restriction systems I use in this chapter are based on their work.

In the field of partial evaluation, Consel and Danvy [21] have described how negative information can be incorporated into a naïvely specialised matcher, thereby achieving effects similar to those described in the present paper. This, however, is achieved by a non-trivial rewrite of the subject program before partial evaluation is applied, thus rendering full automation difficult.

In the case of generalised Partial Computation [33], Takano [109] has presented a transformation technique that exceeds the power of both Turchin's supercompiler and perfect supercompilation. This extra power, however, stems from an unspecified theorem prover that needs to be fed the properties about primitive functions in the language, axioms for the data structures employed in the program under consideration, etc. Later [110], he replaced the theorem prover by the congruence closure algorithm of Nelson and Oppen [75], which allows for the automatic generation of a KMP matcher from a naïvely specialised algorithm when some properties about list structures are provided. In comparison to supercompilation, generalised Partial Computation as formulated by Takano has no concept of generalisation and will therefore terminate only for a small class of programs.

If one abandons simple functional languages and considers logic programming and constraint logic programming, several accounts exist of equivalent transformation power, e.g., Smith and Hickey [102, 40], Lafave and Gallagher [58, 59]. In these frameworks, search and/or constraint solving facilities of the logic language provides the necessary machinery to avoid redundant computations. In this field, great efforts have been made to produce optimal specialisation, and at the same time to ensure termination, see e.g., Leuschel, De Schreye and Martens [65, 66].

Chapter 7

Tupling

In this chapter I will be briefly describe the transformation technique called *tupling*. The purpose of tupling is to eliminate multiple traversals of a data structure (as opposed to deforestation which eliminates intermediate data structures).

Example 26 Consider the following function *average* which computes the average of a list of numbers by dividing their sum with the length of the list.

$$
\begin{aligned}
&\mathit{average}\ xs && = \mathit{sum}\ xs / \mathit{length}\ xs \\
&\mathit{sum}\ [] && = 0 \\
&\mathit{sum}\ (x : xs) && = x + \mathit{sum}\ xs \\
&\mathit{length}\ [] && = 0 \\
&\mathit{length}\ (x : xs) && = 1 + \mathit{length}\ xs
\end{aligned}
$$

This is an inefficient way to compute *average* because I traverse xs twice. It would be better to traverse xs only once, simultaneously adding and counting its elements.

A function which works in the latter way can be obtained by the transformation shown in Figure 7.1. From this I get the new program

$$
\begin{aligned}
&\mathit{av}\ xs && = \mathbf{let}\ \langle u\ ,\ v\rangle := \mathit{sl}\ xs\ \mathbf{in}\ u/v \\
&\mathit{sl}\ [] && = \langle 0\ ,\ 0\rangle \\
&\mathit{sl}\ (x : xs) && = \mathbf{let}\ \langle u\ ,\ v\rangle := \mathit{sl}\ xs\ \mathbf{in}\ \langle x + u\ ,\ 1 + v\rangle \quad .
\end{aligned}
$$

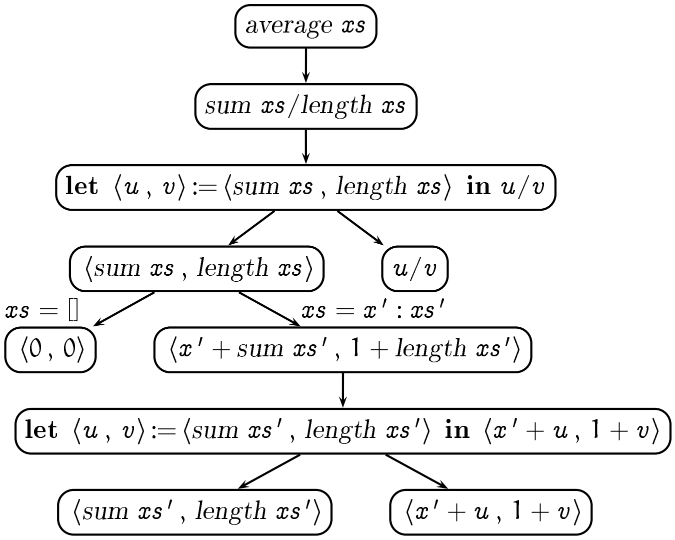

$average\ xs$
$sum\ xs/length\ xs$
$\mathbf{let}\ \langle u\ ,\ v\rangle := \langle sum\ xs\ ,\ length\ xs\rangle\ \mathbf{in}\ u/v$
$\langle sum\ xs\ ,\ length\ xs\rangle$
u/v
$xs = []$
$xs = x' : xs'$
$\langle 0\ ,\ 0\rangle$
$\langle x' + sum\ xs'\ ,\ 1 + length\ xs'\rangle$
$\mathbf{let}\ \langle u\ ,\ v\rangle := \langle sum\ xs'\ ,\ length\ xs'\rangle\ \mathbf{in}\ \langle x' + u\ ,\ 1 + v\rangle$
$\langle sum\ xs'\ ,\ length\ xs'\rangle$
$\langle x' + u\ ,\ 1 + v\rangle$

Figure 7.1: Tranformation using tuples.

This program only traverses xs once, but it may be inefficient due to heap allocation and deallocation of the tuples; in what follows, this latter concern will not be addressed. □

Although the above transformation tree is not unlike the transformation trees seen in earlier chapters, there still remains the important difference that I perform a new kind of abstraction steps where function calls are collected together in a *tuple*; also *several* calls are unfolded (instantiated) simultaneously. The ability to tuple together several calls can have drastic effects on the transformed program.

Example 27 In Section 2.4.2 we encountered the inefficient Fibonacci function:

$$\begin{aligned}
&fib\ 0 &&= 1\\
&fib\ (x+1) &&= fibaux\ x\\
&fibaux\ 0 &&= 1\\
&fibaux\ (x+1) &&= fibaux\ x + fib\ x
\end{aligned}$$

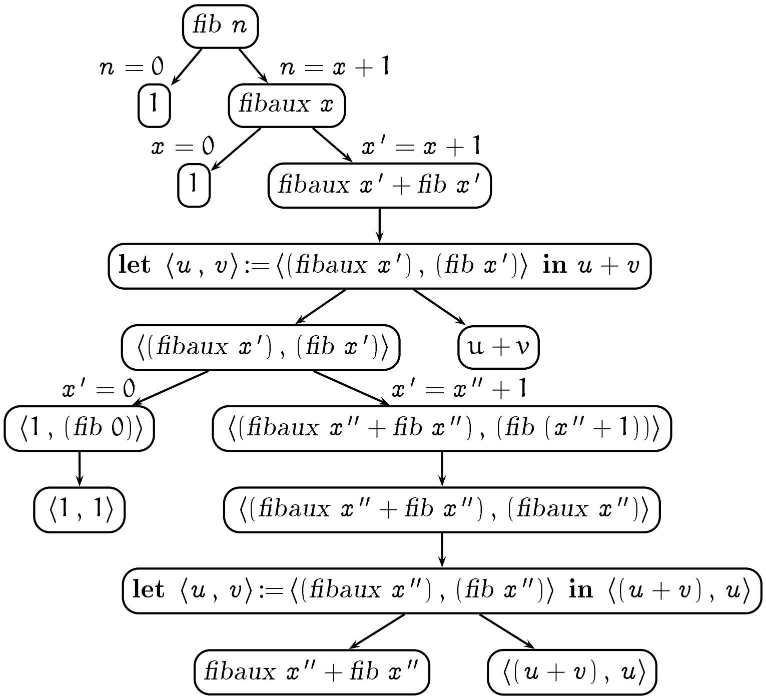

$\mathit{fib}\ n$
$n = 0$
$n = x + 1$
1
$\mathit{fibaux}\ x$
$x = 0$
$x' = x + 1$
1
$\mathit{fibaux}\ x' + \mathit{fib}\ x'$
$\mathbf{let}\ \langle u\,,\ v\rangle := \langle(\mathit{fibaux}\ x')\,,\ (\mathit{fib}\ x')\rangle\ \mathbf{in}\ u + v$
$\langle(\mathit{fibaux}\ x')\,,\ (\mathit{fib}\ x')\rangle$
$u + v$
$x' = 0$
$x' = x'' + 1$
$\langle 1\,,\ (\mathit{fib}\ 0)\rangle$
$\langle(\mathit{fibaux}\ x'' + \mathit{fib}\ x'')\,,\ (\mathit{fib}\ (x'' + 1))\rangle$
$\langle 1\,,\ 1\rangle$
$\langle(\mathit{fibaux}\ x'' + \mathit{fib}\ x'')\,,\ (\mathit{fibaux}\ x'')\rangle$
$\mathbf{let}\ \langle u\,,\ v\rangle := \langle(\mathit{fibaux}\ x'')\,,\ (\mathit{fib}\ x'')\rangle\ \mathbf{in}\ \langle(u + v)\,,\ u\rangle$
$\mathit{fibaux}\ x'' + \mathit{fib}\ x''$
$\langle(u + v)\,,\ u\rangle$

Figure 7.2: Transformation of fib

which can be transformed as shown in Figure 7.2, yielding program

$$\begin{array}{ll}
\mathit{fib}\ 0 & = 1 \\
\mathit{fib}\ (x + 1) & = \mathit{fibaux}\ x \\
\mathit{fibaux}\ 0 & = 1 \\
\mathit{fibaux}\ (x + 1) & = \mathbf{let}\ \langle u\,,\ v\rangle := h\ x\ \mathbf{in}\ u + v \\
h\ 0 & = \langle 1\,,\ 1\rangle \\
h\ (x + 1) & = \mathbf{let}\ \langle u\,,\ v\rangle := h\ x\ \mathbf{in}\ \langle(u + v)\,,\ u\rangle
\end{array}$$

□

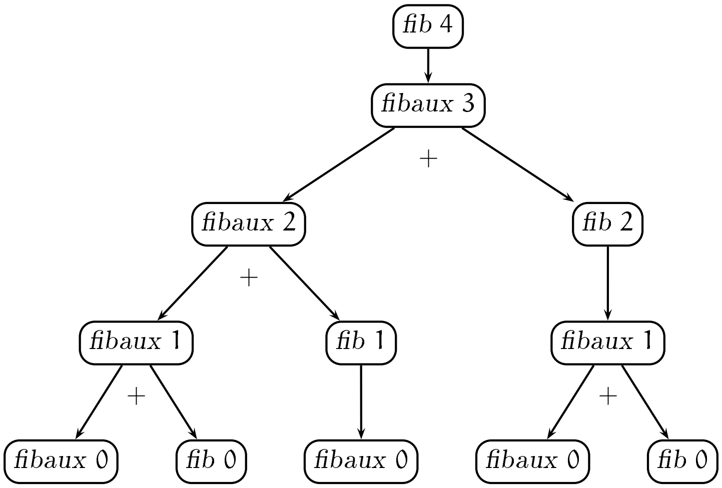


Figure 7.3: Call tree of fib

7.1 Intuition behind Tupling

The techniques called *memoisation* and *tabulation* have traditionally been used to avoid making repeated calls by constructing a lookup table: Each time a function is called with a new parameter, the result of the function call is stored in the table, and the value can then be looked up in the table instead of evaluating each function call over and over again. The aim of tupling is to build this table-storing and looking-up explicitly into the program, so that subsequent evaluation of the transformed program will have the same effect as evaluation with tabulation of the original program. The advantage of the tupling approach is that much of the overhead pertaining to the run-time table management is avoided. The disadvantage is that the tupling approach is more complicated because it attempts to figure out, at compile time, which calls should be stored.

To explain how the tupling transformation works and how tuples are collected, let us look into the notion of *progressive sequence of cuts*. Consider the call tree for *fib* 4 shown in Figure 7.3. For instance, in the above call tree for *fib* 4, you can see that it is possible to compute *fibaux* 3 directly, that is, by performing only an addition, provided you know the values of *fibaux* 2 and

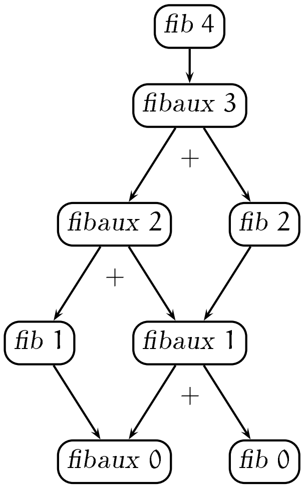


Figure 7.4: Dependency graph

fib 2. Similarly, *fibaux* 2 can be computed directly from *fibaux* 1 and *fib* 1, and so on. The tupling effect is achieved by compressing the call tree into a acyclic directed graph, called the *dependency graph*, as shown in Figure 7.4.

It is possible to express the dependencies of function calls by finding certain regularities in the dependency graphs. Notice that removing any of the set of nodes $\{$*fibaux* 2, *fib* $2\}$, $\{$*fibaux* 1, *fib* $1\}$, or $\{$*fibaux* 0, *fib* $0\}$ divides the graph into two halves, the part above the nodes whose computation relies on the removed nodes (and nothing else), and the part below whose values are required to compute the values of the removed nodes. Since the two parts are disjoint, the sets $\{$*fibaux* 2, *fib* $2\}$, $\{$*fibaux* 1, *fib* $1\}$ and $\{$*fibaux* 0, *fib* $0\}$ are called *cuts*.

The preceding analysis suggests that an efficient way of computing *fib* x is to start from the bottom of the graph and compute tuples $\langle$*fibaux* n, *fib* $n\rangle$, thus working the way to the top. In each step, all the necessary information to perform the addition is present.

The preceding unfold–fold style tupling transformation of *fib* n can be seen as performing this analysis on *abstract* graphs, that is, on *fib* n rather than *fib* 4, and changing the program to build the relevant tuples. The critical property which terminates the construction of the tree in Example 27 is that I move downwards from ⟨*fibaux* x' , *fib* x'⟩ through instantiations to ⟨*fibaux* x'' , *fib* x''⟩, that is, to a renaming of the same tuple. The crux of the matter here is to make sure that each new cut always makes *progress* down the dependency graph, that is, the cut moves downwards. If I can ensure that such a *progressive sequence of cuts* is identified, I can concentrate on finding matching tuples. Unfortunately, one pair of tuples is not enough in general.

Example 28 (Multiple recursive structures) Consider the following program:

```
deepest (LEAF a)       = [ a ]
deepest (BRANCH l r) = if depth l > depth r
                        then deepest l
                        else if depth l < depth r
                             then deepest r
                             else append (deepest l) (deepest r)
depth (LEAF a)         = 0
depth (BRANCH l r)     = 1 + max (depth l) (depth r)
```

This program traverses a data structures (binary trees) that have *several* recursive substructures (in fact, two). For this reason, I need to construct *several* tuples, one for each recursive substructure. The transformation will therefore give something like the tree in Figure 7.5. New program:

```
deepest (LEAF a)       = [ a ]
deepest (BRANCH l r) = let ⟨u , du⟩ := f l; ⟨v , dv⟩ := f r
                         in if du > dv then u
                            else if du < dv then v else append u v
f (LEAF a)             = ⟨[ a ] , 0⟩
f (BRANCH l r)         = let ⟨u , du⟩ := f l; ⟨v , dv⟩ := f r
                           / /if du > dv then u         \                  \
                         in ⟨ ( else if du < dv then v   ) , 1 + max du dv ⟩
                           \ \      else append u v     /                  /
```

□

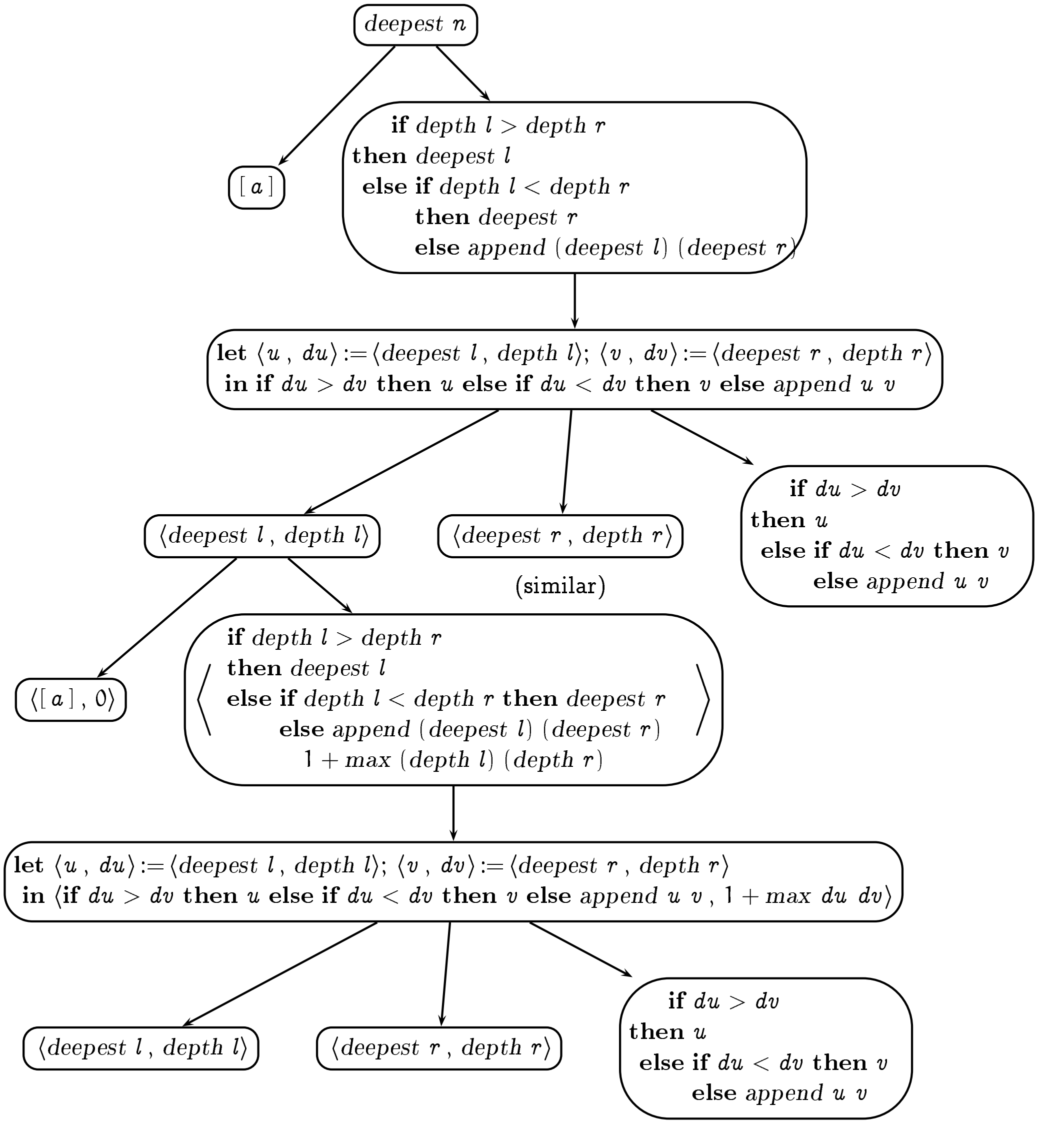

deepest n
[*a*]
if *depth l* > *depth r*
then *deepest l*
else if *depth l* < *depth r*
then *deepest r*
else *append* (*deepest l*) (*deepest r*)
let ⟨*u* , *du*⟩ := ⟨*deepest l* , *depth l*⟩; ⟨*v* , *dv*⟩ := ⟨*deepest r* , *depth r*⟩
in if *du* > *dv* **then** *u* **else if** *du* < *dv* **then** *v* **else** *append u v*
⟨*deepest l* , *depth l*⟩
⟨*deepest r* , *depth r*⟩
(similar)
if *du* > *dv*
then *u*
else if *du* < *dv* **then** *v*
else *append u v*
⟨[*a*] , 0⟩
if *depth l* > *depth r*
then *deepest l*
else if *depth l* < *depth r* **then** *deepest r*
else *append* (*deepest l*) (*deepest r*)
1 + *max* (*depth l*) (*depth r*)
let ⟨*u* , *du*⟩ := ⟨*deepest l* , *depth l*⟩; ⟨*v* , *dv*⟩ := ⟨*deepest r* , *depth r*⟩
in ⟨**if** *du* > *dv* **then** *u* **else if** *du* < *dv* **then** *v* **else** *append u v* , 1 + *max du dv*⟩
⟨*deepest l* , *depth l*⟩
⟨*deepest r* , *depth r*⟩
if *du* > *dv*
then *u*
else if *du* < *dv* **then** *v*
else *append u v*

Figure 7.5: Tupling transformation of '*deepest n*'.

7.2 Further Reading

Semi-automatic techniques for elimination of multiple traversals and repeated function calls have been studied by many authors since the late 1960s, for instance in the work on *tabulation* [7] and *memo-functions* [70]. Besides the already-mentioned Burstall and Darlington paper [13], tupling has been thoroughly investigated by Pettorossi [80], and more recently Chin [14, 16, 17] has studied automatic techniques for tupling of functional programs inspired by these earlier techniques. Related work on tupling of logic programs has been done by Pettorossi and Proietti (e.g., [85]). The way of constructing tuples for each recursive substructure is similar to techniques employed in *conjunctive partial deduction* [26].

The tupling algorithm formulated by Chin is guaranteed to work (and terminate) only for a certain class of programs. This class is defined by a grammar in a way similar to what you have seen for deforestation in Section 4.5, which makes it possible to annotate parts of the program *before* the actual transformation takes place. Such *off-line* techniques speed up the transformation process, since the transformer can happily transform all annotated parts without checking for, for instance, homeomorphic embeddings.

In Chapter 9, I will present a transformation technique that automatically discovers repeated calls in programs — akin to those identified by progressive sequences of cuts — by representing terms as acyclic directed graphs.

Part II

Using Acyclic Directed Graphs

Chapter 8

Dag-based rewriting

In the previous chapters I have discussed various aspects of driving-based program-transformation techniques for functional programs. In this chapter I will lay the foundation for a unified framework based on rewriting *acyclic directed graphs*, as opposed to term rewriting.

8.1 Sharing

It is quite natural to represent terms as acyclic directed graphs (conventionally shortened *dags*). Consider the program $\{\, \mathit{double}\, x = x + x \,\}$ and a term

$$\mathit{double}(\ldots \mathrm{bigcomputation} \ldots) \ ,$$

containing a big computation. Instead of reducing this term to

$$(\ldots \mathit{bigcomputation} \ldots) + (\ldots \mathit{bigcomputation} \ldots) \ ,$$

it is clearly better to represent the terms as dags, as depicted in Figure 8.1. The general idea is that a term t_0 can be represented as a dag Δ_0, which can then be repeatedly rewritten. Each intermediate dag Δ_i represents an intermediate term t_i such that $\mathsf{t} \xrightarrow[\mathsf{trs}(\mathsf{q})]{*} \mathsf{t}_\mathsf{i}$.

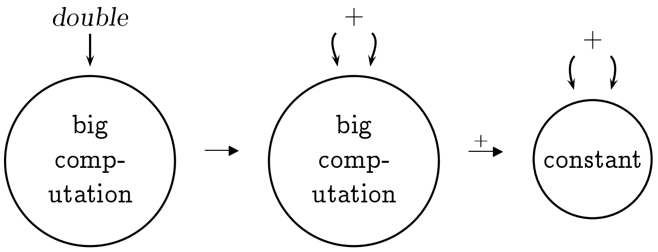


Figure 8.1: Sharing computations by using dags

8.2 Dags and terms

Definition 46 (Marked dag) Let s range over a finite set S, and let α and β range over a set *Addr* of *addresses*. The set of *nodes* is defined by the grammar *Node* $::= s\ \alpha^n$. *Directed graphs* are finite mappings from addresses to nodes or addresses: *Addr* $\Rightarrow$ (*Node* $+$ *Addr*). Any directed graph G induces a *parent-of* binary relation $\multimap \subseteq$ *Addr* $\times$ *Addr* defined as the least relation satisfying

$$\frac{G\alpha = \beta}{G \vdash \alpha \multimap \beta} \qquad \frac{G\alpha = s\ \beta^n}{G \vdash \alpha \multimap \beta_i}\ \forall i \in \{1 \dots n\}\ .$$

A graph G is usually *well formed* (shortened G *w-f*), defined as

$$G\ \textit{w-f} \Leftrightarrow \forall \alpha \in \mathscr{D}G\ (\ G \vdash \alpha \multimap \beta \Rightarrow \beta \in \mathscr{D}G\ )\ .$$

I let Δ range over the set *Dag* of *acyclic directed graphs*, that is, G acyclic $\Leftrightarrow$ $\nexists \alpha \in \mathscr{D}G\ (\ \alpha \overset{+}{\multimap} \alpha\ )$. By $\Delta^{\langle \alpha^n \rangle}$ I denote that the sequence α^n of addresses have been *marked*. By *Mdag* I denote the set of all marked dags.

If an address is mapped to another address (i.e., not to a node), I call both the former address and the mapping an *indirection*, and I define *same-as* relation $\rightsquigarrow$ as the least relation satisfying

$$\frac{}{\Delta \uplus \{\ \alpha \mapsto s\ \beta^n\ \} \vdash \alpha \rightsquigarrow \alpha} \qquad \frac{\Delta \vdash \beta \rightsquigarrow \gamma}{\Delta \uplus \{\ \alpha \mapsto \beta\ \} \vdash \alpha \rightsquigarrow \gamma}\ .$$

As a shorthand, I introduce the relation

$$\frac{\Delta \vdash \alpha \rightsquigarrow \beta \qquad \Delta\beta = N}{\Delta \vdash \alpha \overset{\rightsquigarrow}{=} N}\ .$$

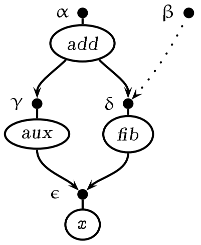
When $A \subseteq \mathit{Addr}$, I let $\Delta_{/A}$ denote the graph Δ restricted to $(\mathscr{D}\Delta) \setminus A$. When $A = \{\, \alpha \,\}$, I simply write $\Delta_{/\alpha}$. I will write "α fresh" to obtain a completely new address (i.e., α has never been used before). □

Remark 5 The same-as relation $\rightsquigarrow$ associates an address with at most one other address.

Remark 6 I will usually consider the marked nodes of a dag to be the *roots* of the dag, although the marked nodes need not be roots in the usual graph-theoretical sense. In general, marked nodes in a dag are used as pointers (or fingers, as it were) into the dag structure.

Example 29 (dag) Let $S = \{\, x \,,\, \mathit{add} \,,\, \mathit{fib} \,,\, \mathit{aux} \,\}$ be a set of symbols and $\mathit{Addr} = \{\, \alpha \,,\, \gamma \,,\, \delta \,,\, \beta \,,\, \epsilon \,,\, \ldots \,\}$ a set of addresses, and consider the dag

$$\Delta \quad = \quad \left\{ \begin{array}{l} \alpha \mapsto \mathit{add}\ \gamma\ \delta \\ \gamma \mapsto \mathit{aux}\ \epsilon \\ \delta \mapsto \mathit{fib}\ \epsilon \\ \beta \mapsto \delta \\ \epsilon \mapsto x \end{array} \right\} .$$



Then $\mathscr{D}\Delta = \{\, \alpha \,,\, \beta \,,\, \gamma \,,\, \delta \,,\, \epsilon \,\}$, and e.g., $\Delta \vdash \beta \overset{\rightsquigarrow}{=} \mathit{fib}\ \ \epsilon$ and $\Delta \vdash \beta \overset{*}{\multimap} \epsilon$. □

Remark 7 As you will see later, indirections are important because they make it possible to reuse already existing parts of a dag by simply overwriting a redex with an indirection. Overwriting a node with an indirection ensures that all parents also benefit from the reuse, and no nodes are duplicated.

From the above example, you can see that I intend to use the set $S = \mathit{Variable} \cup \mathit{Symbol}$ where $\mathit{Symbol} = \mathit{Constructor} \cup \mathit{Function} \cup \mathit{Matcher}$ as the set of symbols that our dags are defined over. I will now clarify the correspondence between such dags and the terms of our term-rewriting language from Chapter 3. It is fairly straightforward to extract a term from a dag.

Definition 47 (Extract) The function $\langle \bullet \rangle_\bullet \in \mathit{Dag} \times \mathit{Addr} \Rightarrow \mathit{Term}$ is defined as

$$\frac{}{\langle \Delta \uplus \{\, \alpha \mapsto \beta \,\} \rangle_\alpha = \langle \Delta \rangle_\beta} \qquad \frac{(\langle \Delta \rangle_{\beta .} = \mathrm{t}.)^n}{\langle \Delta \uplus \{\, \alpha \mapsto \mathrm{s}\ \beta^n \,\} \rangle_\alpha = \mathrm{s}\ \mathrm{t}^n} \ ,$$

where $\mathrm{s} \in \mathit{Symbol}$. When I write just $\langle \Delta \rangle$, I mean a function $\mathit{Addr} \Rightarrow \mathit{Term}$ defined as $\langle \Delta \rangle \overset{\text{def}}{=} \{\, \alpha \mapsto \mathrm{t} \mid \langle \Delta \rangle_\alpha = \mathrm{t} \,\}$. □

Remark 8 The extraction relation defined above is deterministic and "results" in a finite term because any dag is a finite, acyclic map.

Example 30 (Extraction) Consider the dag Δ from Example 29. We have that $\langle \Delta \rangle_\alpha = \mathit{add}\ (\mathit{aux}\ x)\ (\mathit{fib}\ x)$ and $\langle \Delta \rangle_\beta = \mathit{fib}\ x$. □

The opposite translation — from terms to dags — however, depends on how much *sharing* of sub-terms I want to have. Furthermore, the precise formulation of operations on the dag representation of a term will depend on how far I am willing to go to *maintain* sharing of sub-terms. I will define a somewhat complicated *injection* operation on dags to abstract away from such preferences.

Intuitively, an injection does what you would do if you were solving a big jigsaw puzzle: You would form smaller parts of the image by putting together a number of pieces, and you would attach these smaller parts to the large main puzzle.

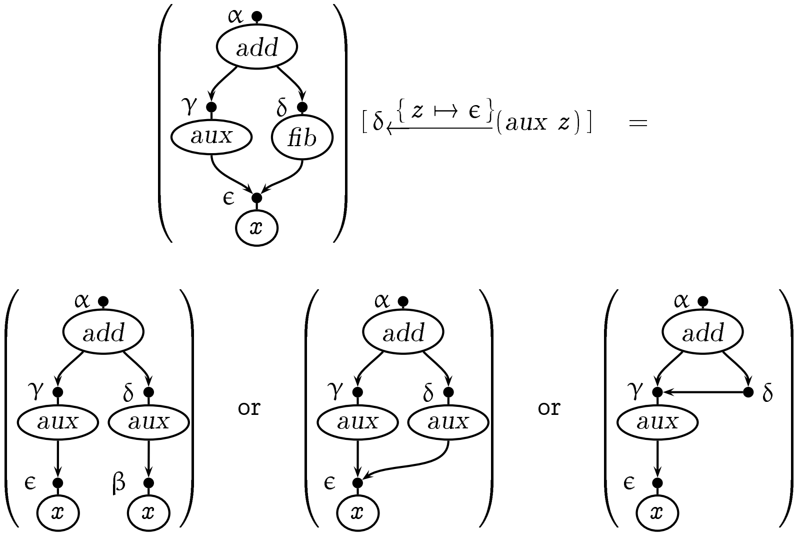
Now, think of a term t as a small image, and think of a dag as the main puzzle. To fit t into a main dag Δ, I need a map θ from the variable in t to addresses in Δ. I will then transform t into a dag Γ such that the variables of t are dangling pointers in Γ. I can then connect the root of Γ to some address α in the main dag Δ, and connect the dangling pointers of Γ to the main dag Δ.

Such an injection of t into Δ at α via θ is denoted $\Delta[\,\alpha \xleftarrow{\theta} \mathrm{t}\,]$. You can see an example of an injection in Figure 8.2.

Definition 48 (Injection) Let $\Delta \in \mathit{Dag}$ be well-formed and $\mathrm{t} \in \mathit{Term}$.

1. A map $\theta \in \mathit{Variable} \Rightarrow \mathit{Addr}$ is *good* iff

$$\mathscr{D}\theta = \mathscr{V}\mathrm{t} \wedge \mathscr{R}\theta \subseteq \mathscr{D}\Delta \ .$$



(a) Duplicating

(b) Standard

(c) Collapsing

Figure 8.2: Example of injection

2. Let $\theta \in \mathit{Variable} \Rightarrow \mathit{Addr}$ be a good map and $\mathscr{V}\mathrm{t} = \langle x^n \rangle$. The t-*yield of* θ, denoted t_θ, is a term defined as

$$t_\theta \overset{\text{def}}{=} t[\,(x_. := \langle\!\langle \Delta \rangle\!\rangle_{(\theta x_.)})^n\,] \enspace .$$

3. Let $\alpha \in \mathit{Addr}$. A good map $\theta \in \mathit{Variable} \Rightarrow \mathit{Addr}$ is α-*admissible* iff

$$\forall \beta \in \mathscr{R}\theta \; \left(\Delta \nvdash \beta \overset{*}{\multimap} \alpha \right) \enspace .$$

4. If θ is an α-admissible map, then the *injection of* t *into* Δ *at* α *via* θ, denoted $\Delta[\,\alpha \overset{\theta}{\hookleftarrow} t\,]$, is defined and is a dag that must satisfy

$$(\Delta[\,\alpha \overset{\theta}{\hookleftarrow} t\,]) = \Delta_{/\alpha} \uplus \Gamma \quad \wedge \quad \langle\!\langle \Delta_{/\alpha} \uplus \Gamma \rangle\!\rangle_\alpha = t_\theta$$

for some $\Gamma \in \mathit{Dag}$. □

Remark 9 The "α-admissible" part in the definition of injection is present to avoid cyclic graphs.

The above says that I only allow injections to extend dags, that is, I require that, as long as I stay inside the domain of the original dag, the same terms will be extracted after the operation, except that α will now materialise into t with the free variables replaced. Note that the injection might have added more than just α to the domain of the original dag.

The definition of the injection operation is abstract in the sense that it does not specify the internal structure of the extension to the original dag; it only specifies what terms I should be able to extract afterwards. This vagueness is deliberate, since it leaves open the decision of how much sharing I want to have in the dags, as mentioned earlier. I will describe three different *realisations* of the injection operation below. I will later relate these realisations to rewriting strategies in the term world.

8.2.1 Standard injection

A straightforward realisation of $\Delta[\,\alpha \overset{\theta}{\hookleftarrow} t\,]$ is shown in Figure 8.3. The auxiliary function *build* converts a term t into dag representation at a target address α. The variables in t are converted into existing addresses in the original dag, thus

$$\Delta \in \mathit{Dag}, \Gamma \in \mathit{Dag}, \alpha \in \mathit{Addr}, \beta \in \mathit{Addr}, t \in \mathit{Term}, x \in \mathit{Variable},$$
$$s \in \mathit{Function} \cup \mathit{Matcher} \cup \mathit{Constructor}, \theta \in \mathit{Variable} \Rightarrow \mathit{Addr}$$

$$\begin{array}{lll}
\Delta[\,\alpha \xleftarrow[\text{STD}]{\theta} t\,] \overset{\text{def}}{=} \textbf{let } \langle \Gamma\,,\ \beta \rangle = \mathit{build}\ t\ \alpha & & \\
\qquad\qquad\quad \textbf{in if } \beta \neq \alpha \textbf{ then } \Delta_{/\alpha} \uplus \{\,\alpha \mapsto \beta\,\} \textbf{ else } \Delta_{/\alpha} \uplus \Gamma & & \\
\text{where}\quad \mathit{build}\ x\ \alpha & \overset{\text{def}}{=} & \langle \varnothing\,,\ (\theta x) \rangle \\
\qquad\quad\ \mathit{build}\ (s\ t^n)\ \alpha & \overset{\text{def}}{=} & \textbf{let } \langle \beta^n \rangle \text{ all fresh} \\
 & & \qquad \langle \Gamma_1\,,\ \alpha_1 \rangle = \mathit{build}\ t_1\ \beta_1 \\
 & & \qquad\qquad \vdots \\
 & & \qquad \langle \Gamma_n\,,\ \alpha_n \rangle = \mathit{build}\ t_n\ \beta_n \\
 & & \textbf{in } \langle (\Gamma_1 \uplus \cdots \uplus \Gamma_n \uplus \{\,\alpha \mapsto s\ \alpha^n\,\})\,,\ \alpha \rangle
\end{array}$$

Figure 8.3: Standard injection

possibly introducing sharing of subdags. More precisely, *build* constructs a new dag while recursively decomposing t. If t has n subterms and a root labelled s, n fresh addresses are chosen and fed to recursive calls to *build* (thus ensuring that the n subterms have been converted into dag representation), and a new node labelled s is created. If t is a variable x, however, nothing is constructed; instead the provided mapping θ indicates which existing address to "substitute" for x. The address representing t can thus be different from the preferred target α, which is why it might be necessary to add an indirection at the top-level definition.

The *standard injection* forms the basis of the so-called *call-by-need* rewriting strategy: Whenever a term is substituted for a variable (i.e., argument to function), the term is shared by all occurrences of the variable. An example of a standard injection is depicted in Figure 8.2(b).

An initial term that one wants to compute is brought into the standard dag world by the following operation.

Definition 49 For any $t \in \mathit{Term}$ with $\mathscr{V} t = \langle x^n \rangle$, I define

$$\mathrm{stddag}(t) \overset{\text{def}}{=} \{\,(\alpha_{\cdot} \mapsto x_{\cdot})^n\,\}\,[\,\alpha_0 \xleftarrow[\text{STD}]{\{\,(x_{\cdot} \mapsto \alpha_{\cdot}\,)^n\,\}} t\,]$$

for some distinct $\alpha_0 \ldots \alpha_n \in \mathit{Addr}$. □

$$\Delta \in \mathit{Dag}, \alpha \in \mathit{Addr}, \beta \in \mathit{Addr}, t \in \mathit{Term}, x \in \mathit{Variable},$$
$$s \in \mathit{Function} \cup \mathit{Matcher} \cup \mathit{Constructor}, \theta \in \mathit{Variable} \Rightarrow \mathit{Addr}$$

$$\Delta[\,\alpha \xleftarrow[\text{DUP}]{\theta} t\,] \overset{\text{def}}{=} \Delta_{/\alpha} \uplus (\mathit{build}\ t\ \alpha)$$

$$\begin{aligned}
\text{where}\quad \mathit{build}\ x\ \alpha &\overset{\text{def}}{=} \textbf{let}\ t = \langle\Delta\rangle_{\theta x}\ \textbf{in}\ \mathit{build}\ t\ \alpha \\
\mathit{build}\ (s\ t^n)\ \alpha &\overset{\text{def}}{=} \textbf{let}\ \langle\beta^n\rangle\ \text{all fresh} \\
&\qquad\quad \Delta_1 = \mathit{build}\ t_1\ \beta_1 \\
&\qquad\qquad\quad \vdots \\
&\qquad\quad \Delta_n = \mathit{build}\ t_n\ \beta_n \\
&\qquad \textbf{in}\ \Delta_1 \uplus \cdots \uplus \Delta_n \uplus \{\,\alpha \mapsto s\ \alpha^n\,\}
\end{aligned}$$

Figure 8.4: Duplicating injection

Lemma 14 *The standard injection shown in Figure 8.3 is a realisation.*

8.2.2 Duplicating injection

The realisation shown in Figure 8.4 never introduces sharing. The auxiliary *build* function constructs a dag representing a specified term at a specified address. The difference from the building function in the *standard-injection* previously described is that when a variable is met, the term represented by the particular sub-dag is extracted and the construction continues. In effect, the existing sub-dag that was to be "substituted" for the variable is duplicated.

The *duplicating injection* forms the basis of the so-called *call-by-name* rewriting strategy: Whenever a term is substituted for a variable (i.e., argument to function), the term is duplicated for each occurrences of the variable. In effect, the constructed dags are always trees. An example of a duplicating injection is depicted in Figure 8.2(a).

Lemma 15 *The duplicating injection shown in Figure 8.4 is a realisation.*

Definition 50 For any $t \in \mathit{Term}$, I define tree(t) to a *tree* Δ such that $\langle\Delta\rangle_\alpha = t$. □

$$\Delta \in \mathit{Dag}, \Gamma \in \mathit{Dag}, \alpha \in \mathit{Addr}, \beta \in \mathit{Addr}, \gamma \in \mathit{Addr}, t \in \mathit{Term}, x \in \mathit{Variable},$$
$$s \in \mathit{Function} \cup \mathit{Matcher} \cup \mathit{Constructor}, \theta \in \mathit{Variable} \Rightarrow \mathit{Addr}$$

```
$\Delta[\,\alpha \xleftarrow[\text{COL}]{\theta} t\,] \overset{\text{def}}{=}$ let $\langle \Gamma , \beta \rangle = \mathit{build}\ t\ \alpha\ \Delta$
                 in if $\beta \neq \alpha$ then $\Delta_{/\alpha} \uplus \{ \alpha \mapsto \beta \}$ else $\Delta_{/\alpha} \uplus \Gamma$
where  $\mathit{build}\ x\ \alpha\ \Delta$          $\overset{\text{def}}{=}$  let $\beta$ suchthat $\Delta \vdash \theta x \rightsquigarrow \beta$ in $\langle \Delta , \beta \rangle$
       $\mathit{build}\ (s\ t^n)\ \alpha\ \Delta_0$   $\overset{\text{def}}{=}$  let $\langle \beta^n \rangle$ all fresh
                                 $\langle \Delta_1 , \alpha_1 \rangle = \mathit{build}\ t_1\ \beta_1\ \Delta_0$
                                          $\vdots$
                                 $\langle \Delta_n , \alpha_n \rangle = \mathit{build}\ t_n\ \beta_n\ \Delta_{n-1}$
                             in if $\exists \beta \in \mathscr{D}\Delta_n$ suchthat $\Delta_n \beta = s\ \gamma^n$
                                                    and $(\Delta_n \vdash \gamma. \rightsquigarrow \alpha.)^n$
                                then $\langle \Delta_n , \beta \rangle$
                                else $\langle (\Gamma_n \uplus \{ \alpha \mapsto s\ \alpha^n \}) , \alpha \rangle$
```

Figure 8.5: Collapsing injection

8.2.3 Collapsing injection

The realisation presented in Figure 8.5 is somewhat the opposite of the duplicating injection, since it will maintain as much sharing as possible. The auxiliary function *build* works like the one used by the *standard injection*, except that a node is not created if a similar one exists. That is, a new node $s\ \alpha^n$ will not be added to a dag Δ at address α if there already exists a node at β such that $\langle\!\langle \Delta \rangle\!\rangle_\beta = \langle\!\langle (\Delta \uplus \{ \alpha \mapsto s\ \alpha^n \}) \rangle\!\rangle_\alpha$. In case such a β exists, function *build* simply returns the dag Δ and the existing address β. Thus, as in the case of the *standard injection*, it might be necessary to add an indirection at the top-level definition to ensure that preferred target address α is defined.

The *collapsing injection* forms the basis of the so-called *collapsed-jungle* rewriting strategy: Every unique term obtainable from a dag is represented by a corresponding unique sub-dag (if one disregards indirections). Thus all occurrences of a particular redex in a term will be shared. An example of a collapsing injection is depicted in Figure 8.2(c).

An initial term that one wants to compute is brought into the dag world by the following operation.

Definition 51 For any $t \in \mathit{Term}$ with $\mathscr{V} t = \langle x^n \rangle$, I define

$$\mathrm{collapse}(t) \overset{\mathrm{def}}{=} \{\, (\alpha_{.} \mapsto x_{.})^n \,\} [\, \alpha_0 \xleftarrow[\textsc{col}]{\{\,(x_{.} \mapsto \alpha_{.})^n\,\}} t \,]$$

for some distinct $\alpha_0 \ldots \alpha_n \in \mathit{Addr}$. □

Lemma 16 *The collapsing injection shown in Figures 8.5 is a realisation.*

8.3 Dag-rewrite system

I will now present a machinery that implements the language defined in Chapter 3 by means of rewriting dags. The machinery is defined as a parameterised small-step operational semantics, in the sense that it is relative to a particular program and a particular injection realisation.

In the following I will assume that I have been given a program q and an injection realisation, that is, I assume that q and $\leftarrow$ are fixed.

Definition 52 (One-step dag reduction) The *one-step dag reduction* relation $\xrightarrow[\mathsf{dag}(q\,\leftarrow)]{} \subseteq \mathit{Mdag} \times \mathit{Mdag}$ is defined as the smallest relation satisfying the inference system in Figure 8.6. □

The inference system is best explained by means of term decompositions (cf. Definition 8).

The relation $\xrightarrow[\mathsf{inner}(q\,\leftarrow)]{}$ relates a marked dag $\Delta^{\langle\alpha\rangle}$ to a marked dag $\Gamma^{\langle\alpha\rangle}$ if $\langle\Delta\rangle_\alpha \xrightarrow[\mathsf{fun}(q)]{} \xrightarrow[\mathsf{trs}(q)]{*} \langle\Gamma\rangle_\alpha$. Put differently, if $\langle\Delta\rangle_\alpha = b[r]$ (for some $b ::= \bullet \mid g\ b\ t^n$ and $r ::= f\ t^n \mid g\ (c\ u^m)\ t^n$), a dag rewrite can take place. The UNFOLD rule applies if $\langle\Delta\rangle_\alpha = f\ t^n$ and the MATCH rule applies if $\langle\Delta\rangle_\alpha = g\ (c\ u^m)\ t^n$. In both cases the right-hand side of the matching function definition in the program q is injected into Δ such that the sub-dags representing the arguments are substituted for the parameters. The DIVE rule applies if $\langle\Delta\rangle_\alpha = g\ (b[r])\ t^n$ and the sub-dag representing $b[r]$ is related to another dag Γ by $\xrightarrow[\mathsf{inner}(q\,\leftarrow)]{}$. If that is the case, a node labelled g is re-injected into Γ to possibly gain more sharing.

The relation $\xrightarrow[\mathsf{outer}(q\,\leftarrow)]{}$ relates a marked dag $\Delta^{\langle\alpha\rangle}$ to a marked dag $\Gamma^{\langle\beta^n\rangle}$ if $\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\,\leftarrow)]{} \Gamma^{\langle\beta^n\rangle}$ or if $\langle\Delta\rangle_\alpha = c\ (\langle\Gamma\rangle_{\beta_{.}})^n$. The relation $\xrightarrow[\mathsf{outer}(q\,\leftarrow)]{}$ thus either

$$\{n, m\} \subseteq \mathbb{N}, t \in \mathit{Term}, \{x, y\} \subseteq \mathit{Variable}, c \in \mathit{Constructor},$$
$$f \in \mathit{Function}, g \in \mathit{Matcher}, \{\Delta, \Gamma\} \subseteq \mathit{Dag}, \{\alpha, \beta, \gamma\} \subseteq \mathit{Addr}$$

$$\text{UNFOLD}\quad \frac{f\ x^n \overset{q}{=} t}{(\Delta \uplus \{\alpha \mapsto f\ \beta^n\})^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\,\leftarrow)]{} (\Delta[\alpha \xleftarrow{\{(x.\mapsto\beta.)^n\}} t])^{\langle\alpha\rangle}}$$

$$\text{MATCH}\quad \frac{g\ (c\ x^m)\ y^n \overset{q}{=} t \qquad\qquad \Delta \vdash \beta_0 \overset{\leadsto}{=} c\ \gamma^m}{(\Delta \uplus \{\alpha \mapsto g\ \beta_0\ \beta^n\})^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\,\leftarrow)]{} (\Delta[\alpha \xleftarrow{\{(x.\mapsto\gamma.)^m\ (y.\mapsto\beta.)^n\}} t])^{\langle\alpha\rangle}}$$

$$\text{DIVE}\quad \frac{\Delta \vdash \beta_1 \leadsto \beta \qquad (\Delta \uplus \{\alpha \mapsto g\ \beta^n\})^{\langle\beta\rangle} \xrightarrow[\mathsf{inner}(q\,\leftarrow)]{} \Gamma^{\langle\beta\rangle}}{(\Delta \uplus \{\alpha \mapsto g\ \beta^n\})^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\,\leftarrow)]{} (\Gamma[\alpha \xleftarrow{\{(x.\mapsto\beta.)^n\}} g\ x^n])^{\langle\alpha\rangle}}$$

$$\text{TRANS}\quad \frac{\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\,\leftarrow)]{} \Gamma^{\langle\alpha\rangle}}{\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{outer}(q\,\leftarrow)]{} \Gamma^{\langle\alpha\rangle}}$$

$$\text{CONS}\quad \frac{}{(\Delta \uplus \{\alpha \mapsto c\ \beta^n\})^{\langle\alpha\rangle} \xrightarrow[\mathsf{outer}(q\,\leftarrow)]{} (\Delta \uplus \{\alpha \mapsto c\ \beta^n\})^{\langle\beta^n\rangle}}$$

$$\text{SEQ}\quad \frac{\Delta \vdash \alpha_0 \leadsto \beta \qquad \Delta^{\langle\beta\rangle} \xrightarrow[\mathsf{outer}(q\,\leftarrow)]{} \Gamma^{\langle\beta^m\rangle}}{\Delta^{\langle\alpha_0\ \alpha^n\rangle} \xrightarrow[\mathsf{dag}(q\,\leftarrow)]{} \Gamma^{\langle\beta^m\ \alpha^n\rangle}}$$

Figure 8.6: Non-strict, small-step dag rewriting

"performs" a dag rewrite or surpasses an outermost constructor, in which case the addresses β^n are the children of α, from left to right.

Finally, the relation $\xrightarrow[\mathsf{dag}(q\leftarrow)]{}$ is responsible for relating dags with several marked addresses: A marked dag $\Delta^{\langle\alpha_0\ \alpha^n\rangle}$ is related to a marked dag $\Gamma^{\langle\beta^m\ \alpha^n\rangle}$ if the difference between Δ and Γ is either a rewrite at a leftmost child of α_0 or if $\langle\Delta\rangle_{\alpha_0} = c\ (\langle\Gamma\rangle_{\beta.})^m$. The relation $\xrightarrow[\mathsf{dag}(q\leftarrow)]{}$ thus mimics outermost-leftmost term-rewriting, a claim I will substantiate below.

That constructors are surpassed may seem a bit strange at first, but, as far as rewriting is concerned, they serve no purpose once they surface, as it were. Intuitively, the rewriting rules maintains a *frontier* of marked addresses that have to be rewritten next in the dag.

Lemma 17 *Relation* $\xrightarrow[\mathsf{dag}(q\leftarrow)]{}$ *is deterministic (for fixed* q *and* $\leftarrow$*)*

PROOF (sketch). By induction and case analysis of the structure of a given dag. □

The determinisity of dag rewriting, as defined here, is important because one can always look at the inference rules to find out what the semantics of a particular program is. Were the choice of the order of evaluation left open, the time consumption of a particular program would be hard to predict (even with a particular injection function in mind).

Dag rewriting and term rewriting, as defined in this thesis, are semantically equivalent. The justification is the following theorem.

Theorem 3 *Let* $t \in$ *Term and* $\Delta^{\langle\alpha\rangle} \in$ *Mdag. If* $t = \langle\Delta\rangle_\alpha$*, then*

$$[\![t]\!]_q = \bigcup_{\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Gamma^{\langle\beta^n\rangle}} \mathscr{O}(\langle\Gamma\rangle_\alpha) \ .$$

8.3.1 Proof of Theorem 3

The proof is rather long, and I will need a couple of auxiliary definitions. The first definition relates an address to all the addresses immediately reachable on the path to the redex.

Definition 53 Given a dag Δ, the *is-context-of* relation $\hookrightarrow \subseteq \mathit{Addr} \times \mathit{Addr}^{\star}$ is defined as the smallest relation satisfying

$$\frac{\Delta \vdash \alpha \overset{\rightsquigarrow}{=} \mathsf{f}\ \beta^n}{\Delta \vdash \alpha \hookrightarrow \langle \alpha \rangle} \qquad\qquad \frac{\Delta \vdash \alpha \overset{\rightsquigarrow}{=} \mathsf{g}\ \beta_0\ \beta^n}{\Delta \vdash \alpha \hookrightarrow \langle \alpha \rangle}$$

$$\frac{\Delta \vdash \alpha \overset{\rightsquigarrow}{=} \mathsf{c}\ \alpha^k\ \beta\ \beta^n \qquad (\llparenthesis \Delta \rrparenthesis_{\alpha.} \in \mathit{Value})^k \qquad \Delta \vdash \beta \hookrightarrow \langle \gamma^m \rangle}{\Delta \vdash \alpha \hookrightarrow \langle \gamma^m\ ,\ \beta^n \rangle}$$

□

Definition 54 Let $t \in \mathit{Term}$, $\Delta \in \mathit{Dag}$ and $\{\,\alpha\,\}\,\alpha^n \subseteq \mathscr{D}\Delta$. I say that t and $\Delta^{\langle \alpha^n \rangle}$ is in *α-bisimulation*, denoted $t \approx_\alpha \Delta^{\langle \alpha^n \rangle}$, when the following holds.

$$t \approx_\alpha \Delta^{\langle \alpha^n \rangle} \qquad \text{iff} \qquad t \xrightarrow[\neg\mathsf{fun}(q)]{*} \llparenthesis \Delta \rrparenthesis_\alpha \quad \wedge \quad \Delta \vdash \alpha \hookrightarrow \langle \alpha^n \rangle \quad .$$

□

PROOF of Theorem 3. I will first prove inclusion from right to left ($\supseteq$).

1. $t = \llparenthesis \Delta \rrparenthesis_\alpha$ [Assumption]
2. $x \in \mathscr{O}(\llparenthesis \Gamma \rrparenthesis_\alpha)$ where $\Delta^{\langle \alpha \rangle} \xrightarrow[\mathsf{dag}(q \leftarrow)]{k} \Gamma^{\langle \beta^n \rangle}$ (for some $k, \Gamma^{\langle \beta^n \rangle}$) [Assumption]
3. $\llparenthesis \Delta \rrparenthesis_\alpha \xrightarrow[\mathsf{trs}(q)]{*} \llparenthesis \Gamma \rrparenthesis_\alpha$ [2 & k×Lemma 19]
4. $t \xrightarrow[\mathsf{trs}(q)]{*} \llparenthesis \Gamma \rrparenthesis_\alpha$ [1 & 3]
5. $x \in [\![t]\!]_q$ [2 & 4 & Definition 7]

The inclusion from left to right ($\subseteq$).

1. $t = \llparenthesis \Delta \rrparenthesis_\alpha$ [Assumption]
2. $x \in [\![t]\!]_q$ [Assumption]
3. $x \in \mathscr{O}t'$ where $t \xrightarrow[\mathsf{fun}(q)]{k} t'$ (for some k, t') [2 & Proposition 1]

Either $k = 0$ or $k > 0$. First he former case:

4. $k = 0$ [Assumption]

5. $x \in \mathscr{O}t$ [3 & 4]

6. $x \in \mathscr{O}\langle\!|\Delta|\!\rangle_\alpha$ [1 & 5]

7. $x \in \bigcup_{\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Gamma^{\langle\beta^n\rangle}} \mathscr{O}(\langle\!|\Gamma|\!\rangle_\alpha)$ [6]

In the latter case:

8. $k > 1$ [Assumption]

9. $t \xrightarrow[\mathsf{fun}(q)]{} u \xrightarrow[\mathsf{fun}(q)]{*} t'$ (for some u) [8 & 3]

10. $\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Delta^{\langle\alpha^n\rangle}$ (for some α^n) [Lemma 28]

11. $\Delta \vdash \alpha \hookrightarrow \langle\alpha^n\rangle$ [*do.*]

12. $t \approx_\alpha \Delta^{\langle\alpha^n\rangle}$ [1 & 11 & Definition 54]

The reduction sequence $u \xrightarrow[\mathsf{fun}(q)]{*} t'$ is unique, so Lemma 21 can now be applied until a term u' for which $t' \xrightarrow[\mathsf{fun}(q)]{*} u'$ is identified:

13. $\Delta^{\langle\alpha^n\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Gamma^{\langle\beta^m\rangle}$ (for some $\Gamma^{\langle\beta^m\rangle}$) [12 & 9 & multiple Lemma 21]

14. $u \xrightarrow[\mathsf{fun}(q)]{*} u' \approx_\alpha \Gamma^{\langle\beta^m\rangle}$ (for some u') [*do.*]

15. $t' \xrightarrow[\mathsf{fun}(q)]{*} u'$ [Assumption]

16. $x \in \mathscr{O}u'$ [3 & 15 & Lemma 2]

17. $u' \xrightarrow[\mathsf{trs}(q)]{*} \langle\!|\Gamma|\!\rangle_\alpha$ [14 & Definition 54]

18. $\mathscr{O}u' \subseteq \mathscr{O}\langle\!|\Gamma|\!\rangle_\alpha$ [17 & Lemma 2]

19. $x \in \bigcup_{\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Gamma^{\langle\beta^n\rangle}} \mathscr{O}(\langle\!|\Gamma|\!\rangle_\alpha)$ [10 & 13 & 18]

□

Lemma 18

$$\forall\gamma \in \mathscr{D}\Delta \left(\langle\!|\Delta[\,\alpha \overset{\theta}{\leftarrow} t\,]|\!\rangle_\gamma = (\langle\!|\Delta_{/\alpha} \uplus \{\,\alpha \mapsto y\,\}|\!\rangle_\alpha)[y := t[\,(x_. := \langle\!|\Delta_{/\alpha}|\!\rangle_{(\theta x_.)})^n\,]\,] \right) \quad ,$$

where y *fresh.*

PROOF. By induction on the structure of Δ w.r.t. γ. □

Definition 55

$$\mathrm{spine}_\alpha\Delta \overset{\text{def}}{=} \begin{cases} \{\,\alpha\,\}, & \text{if } \Delta\alpha = \mathsf{f}\ \beta^n \\ \{\,\alpha\,\} \cup (\mathrm{spine}_{\beta_0}\Delta), & \text{if } \Delta\alpha = \mathsf{g}\ \beta_0\,,\ \beta^n \\ \mathrm{spine}_\beta\Delta, & \text{if } \Delta\alpha = \beta \\ \varnothing, & \text{otherwise.} \end{cases}$$

When $\mathrm{spine}_\alpha\Delta = \{\,\alpha^n\,\}$, I write $\Delta_{/\alpha\text{-spine}}$ for $(\ldots((\Delta_{/\alpha_1})_{/\alpha_2})\ldots)_{/\alpha_n}$. □

Lemma 19 $\Delta^{\langle\alpha_0\,,\,\alpha^n\rangle} \xrightarrow[\mathsf{dag}(q\,\leftarrow)]{} \Gamma^{\langle\gamma^k\rangle} \quad\Rightarrow\quad \forall\alpha \in \mathscr{D}\Delta\ \left(\langle\!|\Delta|\!\rangle_\alpha \xrightarrow[\mathsf{trs}(q)]{*} \langle\!|\Gamma|\!\rangle_\alpha\right).$

PROOF.

1. $\Delta^{\langle\alpha_0\,,\,\alpha^n\rangle} \xrightarrow[\mathsf{dag}(q\,\leftarrow)]{} \Gamma^{\langle\gamma^k\rangle}$ [Assumption]
2. $\Delta \vdash \alpha_0 \rightsquigarrow \beta$ [1 & SEQ (Definition 52)]
3. $\Delta^{\langle\beta\rangle} \xrightarrow[\mathsf{outer}(q\,\leftarrow)]{} \Gamma^{\langle\beta^m\rangle}$ (for some β^m) [*do.*]

Either rule TRANS or rule CONS has been used. In the latter case:

4. $\Delta\beta = \mathsf{c}\ \beta^m$ [Assumption]
5. $\Gamma = \Delta$ [Assumption]
6. $\forall\gamma \in \mathscr{D}\Delta\ (\,\langle\!|\Delta|\!\rangle_\gamma = \langle\!|\Gamma|\!\rangle_\gamma\,)$ [5]

In the former case (TRANS):

7. $\Delta^{\langle\beta\rangle} \xrightarrow[\mathsf{inner}(q\,\leftarrow)]{} \Gamma^{\langle\beta\rangle}$ [Assumption]
8. $\forall\gamma \in \mathscr{D}\Delta\ \left(\langle\!|\Delta|\!\rangle_\gamma \xrightarrow[\mathsf{trs}(q)]{*} \langle\!|\Gamma|\!\rangle_\gamma\right)$ [7 & Lemma 20]

□

Lemma 20

$$\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\,\leftarrow)]{} \Gamma^{\langle\alpha\rangle}$$
$$\Rightarrow$$
$$\exists\Delta'\ \left(\Gamma = \Delta_{/\alpha\text{-}spine} \uplus \Delta' \wedge \forall\gamma \in \mathscr{D}\Delta\ \left(\langle\!|\Delta|\!\rangle_\gamma \xrightarrow[\mathsf{trs}(q)]{*} \langle\!|\Gamma|\!\rangle_\gamma\right)\right) \quad .$$

PROOF. By induction on the depth of the inference tree for $\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\leftarrow)]{} \Gamma^{\langle\alpha\rangle}$. Base cases: Either rule UNFOLD or MATCH was used. I only show the former case.

1. $f\ x^n \overset{q}{=} t$ (for some t, x^n) [Assumption]
2. $\Delta\alpha = f\ \beta^n$ [Assumption]
3. $\Gamma = \Delta_{/\alpha}[\,\alpha \xleftarrow{\{\,(x_.:=\beta_.\,)^n\,\}} t\,]$ [Assumption]
4. $(\Delta_{/\alpha}[\,\alpha \xleftarrow{\{\,(x_.\mapsto\beta_.\,)^n\,\}} t\,]) = \Delta_{/\alpha} \uplus \Delta'$ (for some Δ')
   [$(\Delta_{/\alpha})_{/\alpha} = \Delta_{/\alpha}$ & Definition 48]
5. $\forall\gamma \in \mathscr{D}\Delta\ \big(\,\langle\!\langle\Delta_{/\alpha} \uplus \Delta'\rangle\!\rangle_\gamma = (\langle\!\langle\Delta_{/\alpha} \uplus \{\,\alpha \mapsto y\,\}\rangle\!\rangle_\gamma)[\,y := t[\,(x_. := \langle\!\langle\Delta_{/\alpha}\rangle\!\rangle_{\beta_.}\,)^n\,]\,]\,\big)$
   [Lemma 18]
6. $\forall\gamma \in \mathscr{D}\Delta\ \big(\,\langle\!\langle\Delta\rangle\!\rangle_\gamma = (\langle\!\langle\Delta_{/\alpha} \uplus \{\,\alpha \mapsto y\,\}\rangle\!\rangle_\gamma)[\,y := f\ (x_. := \langle\!\langle\Delta_{/\alpha}\rangle\!\rangle_{\beta_.}\,)^n\,]\,\big)$
   [2 & $\alpha \rightsquigarrow \beta_i$]
7. $\forall\gamma \in \mathscr{D}\Delta\ \Big(\,\langle\!\langle\Delta\rangle\!\rangle_\gamma \xrightarrow[\mathsf{trs}(q)]{*} \langle\!\langle\Delta_{/\alpha} \uplus \Delta'\rangle\!\rangle_\gamma\,\Big)$ [5 & 6 & 1]
8. $\mathrm{spine}_\alpha\Delta = \{\,\alpha\,\}$ [2 & Definition 55]
9. $\Gamma = \Delta_{/\alpha\text{-spine}} \uplus \Delta'$ [3 & 4 & 8]
10. $\forall\gamma \in \mathscr{D}\Delta\ \Big(\,\langle\!\langle\Delta\rangle\!\rangle_\gamma \xrightarrow[\mathsf{trs}(q)]{*} \langle\!\langle\Gamma\rangle\!\rangle_\gamma\,\Big)$ [7 & 3]

Induction case: Rule DIVE have been used.

1. $\Delta\alpha = g\ \beta^n$ [Assumption]
2. $\Delta \vdash \beta_1 \rightsquigarrow \beta$ [Assumption]
3. $\Delta^{\langle\beta\rangle} \xrightarrow[\mathsf{inner}(q\leftarrow)]{} \Delta_1^{\langle\beta\rangle}$ [Assumption]
4. $\Gamma = \Delta_1[\,\alpha \xleftarrow{\{\,(x_.:=\beta_.\,)^n\,\}} g\ x^n\,]$ [Assumption]
5. $\Delta_1 = \Delta_{/\beta\text{-spine}} \uplus \Delta'$ (for some Δ') [3 & induction hypothesis]
6. $\forall\gamma \in \mathscr{D}\Delta\ \Big(\,\langle\!\langle\Delta\rangle\!\rangle_\gamma \xrightarrow[\mathsf{trs}(q)]{*} \langle\!\langle\Delta_1\rangle\!\rangle_\gamma\,\Big)$ [*do.*]

7. $\Gamma = \Delta_{1/\alpha} \uplus \Delta''$ (for some Δ'') [4 & Definition 48]
8. $\llangle \Gamma \rrangle_\alpha = g\ (\llangle \Delta_1 \rrangle_{\beta.})^n$ [*do.*]
9. $\Delta_1\, \alpha = g\ \beta^n$ [$\alpha \notin \mathrm{spine}_\beta \Delta$ & 5 & 1]
10. $\llangle \Delta_1 \rrangle_\alpha = g\ (\llangle \Delta_1 \rrangle_{\beta.})^n$ [9]
11. $\forall \gamma \in \mathscr{D}\Delta_1\ (\ \llangle \Delta_1 \rrangle_\gamma = \llangle \Gamma \rrangle_\gamma\ )$ [10 & 8 & 7]
12. $\mathrm{spine}_\alpha \Delta = \{\, \alpha \,\} \cup \mathrm{spine}_\beta \Delta$ [1 & 2 & Definition 55]
13. $\Gamma = \Delta_{/\alpha\text{-spine}} \uplus \Delta'_{/\alpha} \uplus \Delta''$ [12 & 7 & 5]
14. $\forall \gamma \in \mathscr{D}\Delta \left(\llangle \Delta \rrangle_\gamma \xrightarrow[\mathsf{trs(q)}]{*} \llangle \Gamma \rrangle_\gamma \right)$ [6 & 11]

□

Lemma 21 *Let* $t \approx_\alpha \Delta^{\langle \alpha^n \rangle}$. *If* $t \xrightarrow[\mathsf{fun(q)}]{} u$, *then*

$$\exists \Gamma^{\langle \beta^m \rangle}, t' \left(\Delta^{\langle \alpha^n \rangle} \xrightarrow[\mathsf{dag(q \leftarrow)}]{*} \Gamma^{\langle \beta^m \rangle} \wedge u \xrightarrow[\mathsf{fun(q)}]{*} t' \approx_\alpha \Gamma^{\langle \beta^m \rangle} \right) \quad .$$

PROOF.

1. $t \approx_\alpha \Delta^{\langle \alpha^n \rangle}$ [Assumption]
2. $t \xrightarrow[\mathsf{fun(q)}]{} u$ [Assumption]
3. $t = e[r] \wedge r \xrightarrow[\mathsf{redex(q)}]{} s$ (for some e, r, s) [2]
4. $t \xrightarrow[\neg\mathsf{fun(q)}]{k} \llangle \Delta \rrangle_\alpha$ [1 & Definition 54]
5. $\Delta \vdash \alpha \hookrightarrow \langle \alpha^n \rangle$ [*do.*]
6. $\llangle \Delta \rrangle_\alpha = e'[r'] \wedge r' \xrightarrow[\mathsf{redex(q)}]{} u'$ (for some e', r', u') [3 & 4 & k×Lemma 22]
7. $\Delta^{\langle \alpha^n \rangle} \xrightarrow[\mathsf{dag(q \leftarrow)}]{*} \Gamma^{\langle \beta^m \rangle}$ (for some $\Gamma^{\langle \beta^m \rangle}$) [5 & 6 & Lemma 24]
8. $\llangle \Delta \rrangle_\alpha \xrightarrow[\mathsf{trs(q)}]{*} \llangle \Gamma \rrangle_\alpha$ [*do.*]
9. $\Gamma \vdash \alpha \hookrightarrow \langle \beta^m \rangle$ [*do.*]

10. $t \xrightarrow[\mathsf{trs}(q)]{*} \langle\Gamma\rangle_\alpha$ [4 & 8]

11. $t \xrightarrow[\mathsf{fun}(q)]{*} t' \xrightarrow[\neg\mathsf{fun}(q)]{*} \langle\Gamma\rangle_\alpha$ (for some t') [10 & Lemma 5]

12. $u \xrightarrow[\mathsf{fun}(q)]{*} t' \approx_\alpha \Gamma^{\langle\beta^m\rangle}$ [2 & 11 & 9]

□

Lemma 22

$$r \xrightarrow[\mathsf{redex}(q)]{} u \wedge e[r] \xrightarrow[\neg\mathsf{fun}(q)]{*} t' \Rightarrow \exists e', r', u' \left(t' = e'[r'] \wedge r' \xrightarrow[\mathsf{redex}(q)]{} u' \right) .$$

PROOF. Either $r = f\ t^n$ for some t^n, or $r = g\ (c\ u^m)\ t^n$ for some $t^n\ u^m$. I only show the former case.

1. $r = f\ t^n$ [Assumption]

2. $r \xrightarrow[\mathsf{redex}(q)]{} u$ [Assumption]

3. $e[r] \xrightarrow[\neg\mathsf{fun}(q)]{*} t'$ [Assumption]

I proceed by induction on the structure of e. Base case:

4. $e[r] = r$ [Assumption]

5. $r \xrightarrow[\neg\mathsf{fun}(q)]{*} f\ u^n$ (for some u^n) [4 & 3 & 1]

6. $t' = e[f\ u^n] \wedge f\ u^n \xrightarrow[\mathsf{redex}(q)]{} u'$ (for some u') [5 & 4]

Induction level 1 ($e \in b ::= \bullet \mid g\ b\ t^n$):

7. $e[r] = g\ (b[r])\ t^n$ [Assumption]

8. $g\ (b[r])\ t^n \xrightarrow[\neg\mathsf{fun}(q)]{*} g\ u_0\ u^n$ (for some u_0, u^n) [7 & 3]

9. $b[r] \xrightarrow[\neg\mathsf{fun}(q)]{*} u_0$ [8]

10. $u_0 = b'[r']$ [9 & induction hypothesis]

11. $r' \xrightarrow[\mathsf{redex}(q)]{} u'$ [*do.*]

12. $e[r] \xrightarrow[\neg\mathsf{fun}(q)]{*} g\ (b'[r'])\ u^n$ [11 & 8 & 10]

Induction level 2 ($e[r] = c \ldots$): Similar to level 1. □

Lemma 23 $\langle\Delta\rangle_\alpha = e[r] \wedge \Delta \vdash \alpha \hookrightarrow \langle\alpha^n\rangle \Rightarrow \langle\Delta\rangle_{\alpha_1} = b[r]$ *for some* $b ::= \bullet \mid g\ b\ t^n$ *(as defined as in Definition 8).*

PROOF (sketch). By induction on e.

Lemma 24

$$\begin{gathered} \langle\Delta\rangle_\alpha = e[r] \wedge r \xrightarrow[\mathsf{redex}(q)]{} u \wedge \Delta \vdash \alpha \hookrightarrow \langle\alpha^n\rangle \\ \Rightarrow \\ \Delta^{\langle\alpha^n\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{+} \Gamma^{\langle\beta^k\rangle} \wedge \langle\Delta\rangle_\alpha \xrightarrow[\mathsf{trs}(q)]{*} \langle\Gamma\rangle_\alpha \wedge \Gamma \vdash \alpha \hookrightarrow \langle\beta^k\rangle \quad . \end{gathered}$$

PROOF.

1. $r \xrightarrow[\mathsf{redex}(q)]{} u$ [Assumption]
2. $\langle\Delta\rangle_\alpha = e[r]$ [Assumption]
3. $\Delta \vdash \alpha \hookrightarrow \langle\alpha^n\rangle$ [Assumption]
4. $\langle\Delta\rangle_{\alpha_1} = b[r]$ (for some $b ::= \bullet \mid g\ b\ t^n$) [2 & 3 & Lemma 23]
5. $\Delta \vdash \alpha_1 \rightsquigarrow \beta$ (for some β) [Δ wf.]
6. $\Delta \vdash \beta \rightsquigarrow \beta$ [5]
7. $\langle\Delta\rangle_\beta = b[r]$ [5 & 4]
8. $\Delta^{\langle\beta\rangle} \xrightarrow[\mathsf{inner}(q\leftarrow)]{} \Gamma^{\langle\beta\rangle}$ (for some Γ) [7 & 6 & 1 & Lemma 25]
9. $\Gamma = \Delta_{/\beta\text{-spine}} \uplus \Delta'$ (for some Δ') [8 & Lemma 20]
10. $\forall\gamma \in \mathscr{D}\Delta \left(\langle\Delta\rangle_\gamma \xrightarrow[\mathsf{trs}(q)]{*} \langle\Delta_{/\beta\text{-spine}} \uplus \Delta'\rangle_\gamma \right)$ [*do.*]
11. $\Gamma^{\langle\alpha^n\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Gamma^{\langle\gamma^m\rangle}$ (for some γ^m) [3 & 5 & 9 & 10 & Lemma 26]
12. $\Gamma \vdash \alpha \hookrightarrow \langle\gamma^m\rangle$ [*do.*]
13. $\Delta^{\langle\alpha^n\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{} \Gamma^{\langle\alpha^n\rangle}$ [8 & 5 & 9 & TRANS+SEQ (Definition 52)]
14. $\Delta^{\langle\alpha^n\rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{+} \Gamma^{\langle\gamma^m\rangle}$ [13 & 11]
15. $\langle\Delta\rangle_\alpha \xrightarrow[\mathsf{trs}(q)]{*} \langle\Gamma\rangle_\alpha$ [10]

□

Lemma 25

$$\Delta \vdash \alpha \rightsquigarrow \alpha \wedge \langle\Delta\rangle_\alpha = b[r] \wedge r \xrightarrow[\mathsf{redex}(q)]{} u \quad\Rightarrow\quad \exists\Gamma \left(\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\leftarrow)]{} \Gamma^{\langle\alpha\rangle} \right) \quad .$$

PROOF. Either $r = f\ t^n$ for some t^n, or $r = g\ (c\ u^m)\ t^n$ for some $t^n\ u^m$. I only show the former case.

1. $\Delta \vdash \alpha \rightsquigarrow \alpha$ [Assumption]
2. $r \xrightarrow[\mathsf{redex}(q)]{} u$ [Assumption]
3. $\langle\Delta\rangle_\alpha = b[r]$ [Assumption]

I proceed by induction on the structure of b. Base case:

4. $b[r] = r$ [Assumption]
5. $r = f\ t^n$ [Assumption]
6. $f\ x^n \overset{q}{=} t$ (for some x^n, t) [5 & 2]
7. $\Delta\alpha = f\ \beta^n$ (for some β^n) [1 & 3 & 4 & 5]
8. $\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\leftarrow)]{} \Gamma^{\langle\alpha\rangle}$ (for some Γ) [6 & 7 & UNFOLD (Definition 52)]

Induction case:

9. $b[r] = g\ (b'[r])\ t^{n-1}$ [Assumption]
10. $\Delta\alpha = g\ \beta^n$ (for some β^n) [9 & 1 & 3]
11. $\Delta\beta_1 \rightsquigarrow \beta$ (for some β) [Δ wf]
12. $\Delta\beta \rightsquigarrow \beta$ [*do.*]
13. $\langle\Delta\rangle_\beta = b'[r]$ [9 & 10 & 11]
14. $\Delta^{\langle\beta\rangle} \xrightarrow[\mathsf{inner}(q\leftarrow)]{} \Delta_1^{\langle\beta\rangle}$ (for some Δ_1) [12 & 13 & 2 & induction hypothesis]
15. $\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q\leftarrow)]{} \Gamma^{\langle\alpha\rangle}$ (for some Γ) [10 & 11 & 14 & DIVE]

□

Lemma 26

$$\Delta \vdash \alpha_0 \hookrightarrow \langle \alpha^n \rangle \wedge \Delta \vdash \alpha_1 \rightsquigarrow \beta_0 \wedge \Gamma = \Delta_{/\beta_0\text{-}spine} \uplus \Delta' \wedge \forall \gamma \in \mathscr{D}\Delta \left(\llparenthesis \Delta \rrparenthesis_\gamma \xrightarrow[\mathsf{trs(q)}]{*} \llparenthesis \Gamma \rrparenthesis_\gamma \right)$$
$$\Downarrow$$
$$\exists \gamma^m \left(\Gamma^{\langle \alpha^n \rangle} \xrightarrow[\mathsf{dag(q \leftarrow)}]{*} \Gamma^{\langle \gamma^m \rangle} \wedge \Gamma \vdash \alpha_0 \hookrightarrow \langle \gamma^m \rangle \right) \quad .$$

PROOF.

1. $\Delta \vdash \alpha_1 \rightsquigarrow \beta_0$ [Assumption]
2. $\Delta \vdash \alpha_0 \hookrightarrow \langle \alpha^n \rangle$ [Assumption]
3. $\Gamma = \Delta_{/\beta_0\text{-spine}} \uplus \Delta'$ [Assumption]
4. $\forall \gamma \in \mathscr{D}\Delta \left(\llparenthesis \Delta \rrparenthesis_\gamma \xrightarrow[\mathsf{trs(q)}]{*} \llparenthesis \Gamma \rrparenthesis_\gamma \right)$ [Assumption]
5. $\Gamma^{\langle \alpha_1 \rangle} \xrightarrow[\mathsf{dag(q \leftarrow)}]{*} \Gamma^{\left\langle \beta^k \right\rangle}$ (for some β^k) [Lemma 28]
6. $\Gamma \vdash \alpha_1 \hookrightarrow \left\langle \beta^k \right\rangle$ [*do.*]

I proceed by induction on the structure of Δ w.r.t. α_0. Base case ($\Delta \vdash \alpha_0 \overset{\rightsquigarrow}{=} g \ldots$ is similar):

7. $\Delta \vdash \alpha_0 \overset{\rightsquigarrow}{=} f\ \delta^j$ (for some δ^j) [Assumption]
8. $\Delta \vdash \alpha_0 \hookrightarrow \langle \alpha_0 \rangle$ [7 & Definition 53]
9. $\alpha_0 = \alpha_1$ [8 & 2]
10. $\Gamma^{\langle \alpha^n \rangle} \xrightarrow[\mathsf{dag(q \leftarrow)}]{*} \Gamma^{\langle \gamma^m \rangle} \wedge \Gamma \vdash \alpha_0 \hookrightarrow \langle \gamma^m \rangle$ (for some γ^m) [9 & 5 & 6]

Induction case ($\Delta \vdash \alpha_0 \overset{\rightsquigarrow}{=} c \ldots$):

11. $\Delta \vdash \alpha_0 \overset{\rightsquigarrow}{=} c\ \delta^j\ \gamma_0\ \alpha_{\ell+1}\ \ldots\ \alpha_n$ (for some $\ell \in \{1 \ldots n\}, \delta^j, \gamma_0$) [2 & Definition 53]
12. $(\llparenthesis \Delta \rrparenthesis_{\delta.} \in \mathit{Value})^j$ [*do.*]
13. $\Delta \vdash \gamma_0 \hookrightarrow \left\langle \alpha^\ell \right\rangle$ [2 & 11]
14. $\Gamma \vdash \alpha_0 \overset{\rightsquigarrow}{=} c\ \delta^j\ \gamma_0\ \alpha_{\ell+1}\ \ldots\ \alpha_n$ [3 & 11]
15. $(\llparenthesis \Gamma \rrparenthesis_{\delta.} \in \mathit{Value})^j$ [3 & 12]

16. $\Gamma^{\langle \alpha^{\ell} \rangle} \xrightarrow[\mathsf{dag}(q \leftarrow)]{*} \Gamma^{\langle \gamma^{m} \rangle}$ (for some γ^{m}) [13 & 1 & 3 & 4 & induction hyp.]

17. $\Gamma \vdash \gamma_0 \hookrightarrow \langle \gamma^{m} \rangle$ [*do.*]

18. $\Gamma^{\langle \alpha^{n} \rangle} \xrightarrow[\mathsf{dag}(q \leftarrow)]{*} \Gamma^{\langle \gamma^{m}, \alpha_{\ell+1}, \ldots, \alpha_n \rangle}$ [16 & Lemma 27]

Either $m = 0$ or $m > 0$. The latter case is simple:

19. $m > 0$ [Assumption]

20. $\Gamma \vdash \alpha_0 \hookrightarrow \langle \gamma^{m}, \alpha_{\ell+1}, \ldots \alpha_n \rangle$ [19 & 17 & 14 & 15 & Definition 53]

The former case:

21. $m = 0$ [Assumption]

22. $\langle \Gamma \rangle_{\gamma_0} \in \mathit{Value}$ [21]

23. $\Gamma^{\langle \alpha_{\ell+1}, \ldots, \alpha_n \rangle} \xrightarrow[\mathsf{dag}(q \leftarrow)]{*} \Gamma^{\langle \beta^{k}, \alpha_{\ell+i}, \ldots, \alpha_n \rangle}$ (for some i, β^{k}) [$i\times$ Lemma 28]

24. $\Gamma \vdash \alpha_{\ell} \hookrightarrow \langle \beta^{k} \rangle$ [*do.*]

25. $\langle \Gamma \rangle_{\alpha_{\ell+1}}, \ldots, \langle \Gamma \rangle_{\alpha_{\ell+i-1}} \in \mathit{Value}$ [*do.*]

26. $\Gamma \vdash \alpha_0 \hookrightarrow \langle \beta^{k}, \alpha_{\ell+i}, \ldots, \alpha_n \rangle$
[14 & 15 & 22 & 25 & 24 & Definition 53]

□

Lemma 27 $\Delta^{\langle \alpha_0 \rangle} \xrightarrow[\mathsf{dag}(q \leftarrow)]{} \Gamma^{\langle \beta^{m} \rangle} \Rightarrow \forall \alpha^{n} \left(\Delta^{\langle \alpha_0, \alpha^{n} \rangle} \xrightarrow[\mathsf{dag}(q \leftarrow)]{} \Gamma^{\langle \beta^{m}, \alpha^{n} \rangle} \right).$

PROOF.

1. $\Delta^{\langle \alpha_0 \rangle} \xrightarrow[\mathsf{dag}(q \leftarrow)]{} \Gamma^{\langle \beta^{m} \rangle}$ [Assumption]

2. $\Delta \vdash \alpha_0 \rightsquigarrow \beta$ [Assumption]

3. $\Delta^{\langle \beta \rangle} \xrightarrow[\mathsf{outer}(q \leftarrow)]{} \Gamma^{\langle \beta^{m} \rangle}$ [1 & 2 & SEQ]

4. $\Delta^{\langle \alpha_0, \alpha^{n} \rangle} \xrightarrow[\mathsf{dag}(q \leftarrow)]{} \Gamma^{\langle \beta^{m}, \alpha^{n} \rangle}$ [3 & SEQ]

□

Lemma 28 $\forall \Delta \in \mathit{Dag}, \alpha \in \mathscr{D}\Delta : \exists \beta^m \in \mathscr{D}\Delta$:

$$\begin{gathered} \Delta^{\langle \alpha \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Delta^{\langle \beta^m \rangle} \\ \wedge \\ \Delta \vdash \alpha \hookrightarrow \langle \beta^m \rangle \wedge (m = 0 \Rightarrow \langle\!|\Delta|\!\rangle_\alpha \in \mathit{Value}) \end{gathered} \quad .$$

PROOF. By induction on the structure of Δ. If $\Delta \vdash \alpha \overset{\rightsquigarrow}{=} f\ \alpha^n$ or $\Delta \vdash \alpha \overset{\rightsquigarrow}{=} g\ \alpha^n$ (for some α^n), then trivially

$$\Delta^{\langle \alpha \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Delta^{\langle \alpha \rangle} \wedge \Delta \vdash \alpha \hookrightarrow \langle \alpha \rangle \quad .$$

Otherwise, let $\Delta \vdash \alpha \overset{\rightsquigarrow}{=} c\ \alpha^n$ (for some α^n). By rules SEQ and CONS in Definition 52 it follows that $\Delta^{\langle \alpha \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{} \Delta^{\langle \alpha^n \rangle}$. By the induction hypothesis, there is a maximal $k \in \{0 \ldots n\}$ such that

$$\forall i \in \{1 \ldots k\} \left(\Delta^{\langle \alpha_i \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Delta^{\langle\rangle} \wedge \langle\!|\Delta|\!\rangle_{\alpha_i} \in \mathit{Value} \right) \quad . \tag{8.1}$$

By k times application of Lemma 27,

$$\Delta^{\langle \alpha \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{} \Delta^{\langle \alpha^n \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Delta^{\langle \alpha_{k+1}, \ldots, \alpha_n \rangle} \quad . \tag{8.2}$$

If $k = n$, then by (8.1),

$$\Delta^{\langle \alpha \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Delta^{\langle\rangle} \wedge \langle\!|\Delta|\!\rangle_\alpha \in \mathit{Value} \quad .$$

Otherwise ($k < n$), by maximality of k and the induction hypothesis

$$\Delta^{\langle \alpha_{k+1} \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Delta^{\langle \beta^m \rangle} \wedge \Delta \vdash \alpha_{k+1} \hookrightarrow \langle \beta^m \rangle \quad . \tag{8.3}$$

By $\Delta \vdash \alpha \overset{\rightsquigarrow}{=} c\ \alpha^n$ and (8.1), also $\Delta \vdash \alpha \hookrightarrow \langle \beta^m , \alpha_{k+2} , \ldots , \alpha_n \rangle$, and thus by (8.2) and Lemma 27,

$$\Delta^{\langle \alpha \rangle} \xrightarrow[\mathsf{dag}(q\leftarrow)]{*} \Delta^{\langle \beta^m , \alpha_{k+2} , \ldots , \alpha_n \rangle} \wedge \Delta \vdash \alpha \hookrightarrow \langle \beta^m , \alpha_{k+2} , \ldots , \alpha_n \rangle \quad .$$

□

Intuitively, each time a dag rewrite rewrites an outermost-leftmost redex r into a contractum, it might also rewrite identical redexes to the right of r. This relationship between dag rewriting and term rewriting can be made more precise.

Definition 56 $t \xrightarrow[\mathsf{fun+(q)}]{} t' \Leftrightarrow \exists u, r, r' \left(\begin{array}{ll} & r \xrightarrow[\mathsf{redex(q)}]{} r' \\ \wedge & t = u[x{:=}r] \\ \wedge & t' = u[x{:=}r'] \\ \wedge & x \in \mathscr{V}u \\ \wedge & \text{the leftmost } x \text{ is to the} \\ & \text{left of any redex in } u \end{array} \right)$ □

Lemma 29 $\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{dag(q\leftarrow)}]{} \Gamma^{\langle\beta^{n}\rangle} \Rightarrow \langle\!|\Delta|\!\rangle_{\alpha} \xrightarrow[\mathsf{fun+(q)}]{} \langle\!|\Gamma|\!\rangle_{\alpha}$.

PROOF (sketch). By induction on the structure of $\langle\!|\Delta|\!\rangle_{\alpha}$, using Definition 48. □

8.4 Injection flavours

Having left open the choice of which injection realisation to use when rewriting dags, a natural question is: Which one is better? Or to put it differently, which one is more efficient?

There are at least two ways of measuring efficiency, namely in time and space usage.[1] In terms of space usage, it is obvious that the more sharing, the less asymptotic memory consumption. In terms of time usage, it is also obvious that the more sharing, the more computations can be shared (or done "in parallel", as it were), thus saving computation time. The problem with this line of thought is that there is an *administrative overhead* in maintaining full sharing, as seen in the collapsing-injection realisation.

In lieu of the separation between the operational semantics and the underlying realisation of the graph machinery, it is therefore reasonable to divide the time spent for any computation into *rewriting steps* and *administrative steps*. Figure 8.7 clarifies the typical relationship between the three injection realisations presented earlier. From a term-rewriting perspective, it might be surprising to see that simple substitution induces more administration than call-by-need, but as you can see from the duplicating realisation (Figure 8.4), it is quite unnatural to emulate term-based substitution in the dag world.

The exact relationship between the three strategies depends, of course, on the program at hand. For instance, if every right-hand side in the program

[1] The cache behaviour of a computation has a significant effect on the time it takes to perform the computation, so these matters are not unrelated.

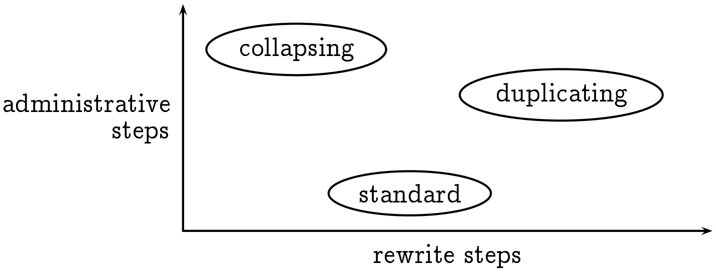


Figure 8.7: Rewriting vs. administrative steps

is linear, the standard and duplicating injections will give the same asymptotic behaviour. Comparing standard and collapsing injections, the asymptotic behaviour can be exponentially better for some programs when the collapsing injection is used, as illustrated by the following example.

Example 31 Assume that q is the Fibonacci Number program

$$\begin{array}{ll}
\textbf{data}\ \mathrm{Nat} & = \mathsf{0} \mid \mathsf{s}\ \mathrm{Nat} \\
\mathit{fib}\ \mathsf{0} & = \mathsf{s}\ \mathsf{0} \\
\mathit{fib}\ (\mathsf{s}\ x) & = \mathit{aux}\ x \\
\mathit{aux}\ \mathsf{0} & = \mathsf{s}\ \mathsf{0} \\
\mathit{aux}\ (\mathsf{s}\ y) & = \mathit{add}\ (\mathit{fib}\ (\mathsf{s}\ y))\ (\mathit{fib}\ y) \\
\mathit{add}\ \mathsf{0}\ y & = y \\
\mathit{add}\ (\mathsf{s}\ x)\ y & = \mathsf{s}\ (\mathit{add}\ x\ y)\ .
\end{array}$$

If $\leftarrow$ is the collapsing injection, any marked dag $\Delta^{\langle\alpha_1\rangle}$ for which $\langle\Delta\rangle_{\alpha_1} = \mathit{fib}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{0})))$ will be rewritten as depicted in Figure 8.8. In general, using the collapsing rewrite strategy w.r.t. the above program above avoids rewriting an exponential number of redexes. □

Rewriting strategies similar to the one I obtain by using collapsing injections has been advocated in the literature, but in practice it seems that the administrative overhead is simply too burdensome, compared to benefits of sharing redexes that would not naturally have been shared by the standard injection. I therefore propose a feasible compromise in the next chapter: Optimise programs

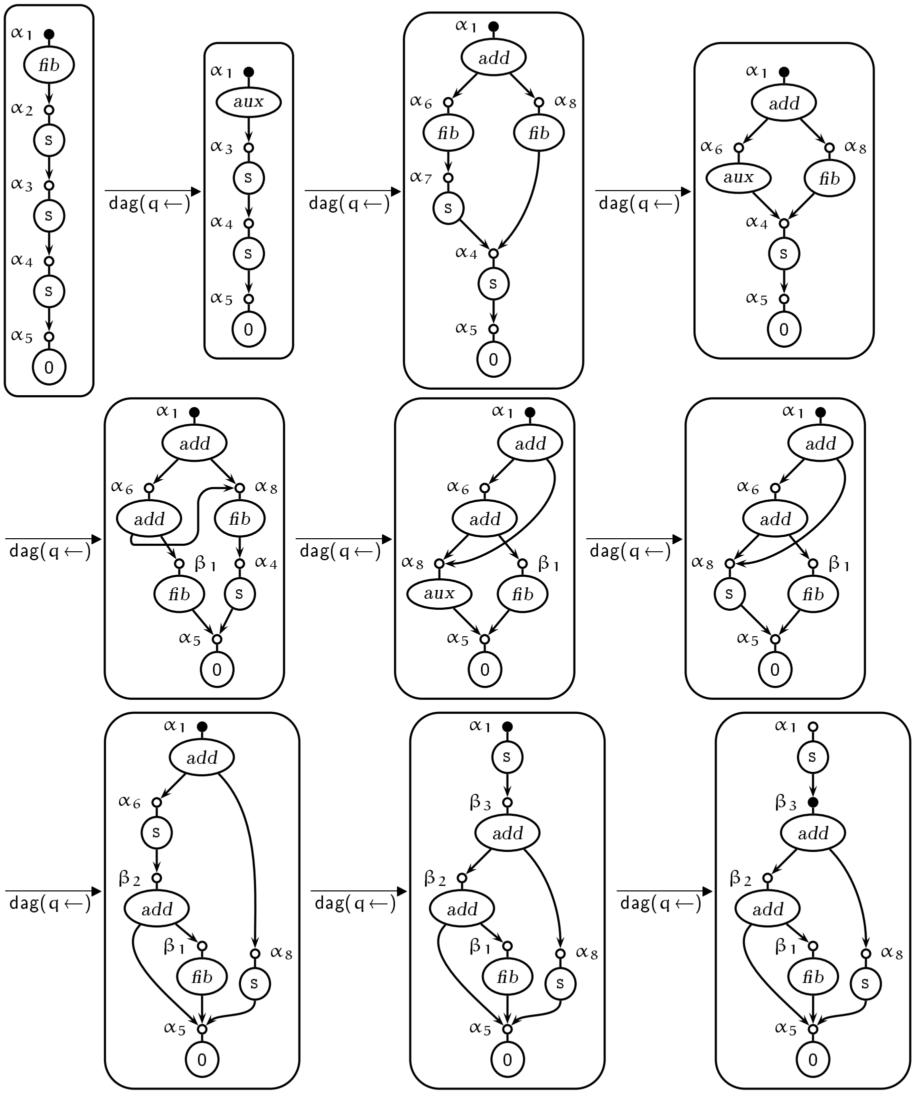


Figure 8.8: Rewriting using collapsing injection

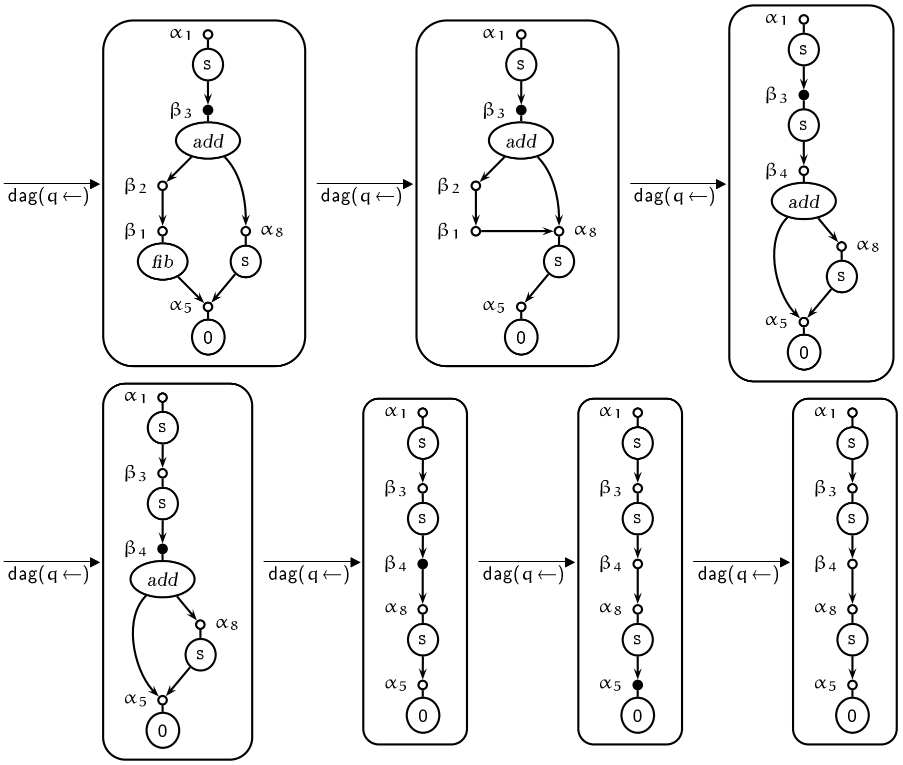


Figure 8.9: Rewriting using collapsing injection (cont.)

at compile time using the collapsing injection, but use standard injections at run time.

8.5 Summary and Related work

I have shown that, for a small functional programming language, any dag-rewriting implementation which obeys a very reasonable *injection* rule will lead to the same semantics. I have given three examples of such implementations, one emulating call-by-need, one emulating call-by-name and one similar to collapsed-jungle evaluation.

Wadsworth [122] invented call-by-need for the pure λ-calculus, and proved that normal-order (call-by-need) graph reduction is at least as efficient as normal-order term reduction for a certain subset of graphs representing λ-terms, and he devised an algorithm for performing normal-order reduction. The presence of bound variables in the λ-calculus, however, forced Wadsworth to focus mainly on the problems arising from cycles introduced by free variables, and he did not consider more aggressive strategies like fully collapsed dags. Hoffmann and Plump [43], the main source of inspiration for the present formulation of dag rewriting, have proved that term rewrite systems could be translated into hypergraph replacement systems. They define the notion of fully-collapsed jungles in terms of morphisms on graphs, and they show uniqueness of such fully-collapsed graphs. Instead of avoiding to build already-present nodes in a dag, they perform separate *fold*-morphism on the dags to ensure that they are fully collapsed. Their main focus, however, is on showing that confluence and termination is preserved for a large class of term rewrite systems. The idea of not building already-present node can be traced back to Sassa and Goto [93], who described what is usually called *hash consing.* Kahrs [52] have later described how hash consing can be used to implement fully-collapsed jungles.

Compared to other graph-rewriting approaches, the merit of the dag machinery presented in this chapter is that the machinery is directly implementable (e.g., no category theory) but at the same time it is highly parameterised. Moreover, there is no focus on normal forms, since the semantics is based on observational behaviour.

Chapter 9

Dag-based program transformation

In this chapter I will present a unified supercompilation framework which can perform most of the optimisations seen in Part I. The framework is based on dag rewriting, as described in the previous chapter. I will show that most of the existing techniques are instances of this unified framework. More importantly, under certain conditions, the framework makes it possible to guarantee that a transformed program is at least as efficient as the original.

As I have described in Part I, previous transformation methods have used call-by-name unfolding, possibly optimised by marking "dangerous" sub-terms as non-unfoldable to avoid duplication. Instead, I will here directly address the root of evil by building sharing into program transformation.

Like previous transformation methods, the dag-based transformer has three conceptual phases. First, a possibly infinite model of the computations of a term w.r.t. a program is constructed. Second, the model is pruned to make it finite. Third, a new program is extracted from the finite model.

9.1 Dag driving

Like in deforestation and supercompilation, a transformation trace tree is constructed by *driving* a term w.r.t. a program. Again, driving the program means to speculatively execute non-ground terms. Towards this end, I make two modifications to the small-step dag-rewriting rules. The first simply allows variables in terms.

Definition 57 (Deterministic unfolding) The relation $\xrightarrow[\mathsf{dagx}(q \leftarrow)]{} \subseteq \mathit{Mdag} \times \mathit{Mdag}$ is defined by adding the following rule to the inference system in Figure 8.6.

$$\text{VAR}\ \frac{\Delta\alpha = x}{\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{outer}(q \leftarrow)]{} \Delta^{\langle\rangle}}\ .$$

□

The new relation $\xrightarrow[\mathsf{dagx}(q \leftarrow)]{}$ is still deterministic, but it allows rewriting steps to ignore what corresponds to uninstantiated parts to the left of a redex. With the relation $\xrightarrow[\mathsf{dagx}(q \leftarrow)]{}$, I can rewrite non-ground terms as long as I do not run into redexes of the form $g\ x\ t^n$. To rewrite such redexes, I need to *speculatively* try out all possible forms of values of x, according to the definition of g, as seen in chapters on deforestation and supercompilation. In the case of dag-based rewriting, however, driving can be formulated simply as outside-in rewriting together with an *instantiation* rule for variables.

Definition 58 (Instantiation) The relation $\xrightarrow[\mathsf{drive}(q \leftarrow)]{} \subseteq \mathit{Mdag} \times \mathit{Mdag}$ is defined as $\xrightarrow[\mathsf{dagx}(q \leftarrow)]{}$, but with the additional rule

$$\text{INST}\ \frac{\begin{array}{ccc} & \Delta\alpha = g\ \beta_0\ \beta^n & \Delta \vdash \beta_0 \rightsquigarrow \beta \\ \Delta\beta = x & g\ (c\ x^m)\ y^n \overset{q}{=} t & \gamma^m \text{ and } x^m \text{ fresh} \end{array}}{\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{inner}(q \leftarrow)]{} (\Delta_{/\beta} \uplus \{ (\beta \mapsto c\ \gamma^m)(\gamma_. \mapsto x_.)^m \})^{\langle\alpha\rangle}}\ .$$

□

The relation $\xrightarrow[\mathsf{drive}(q \leftarrow)]{}$ is non-deterministic: When it encounters a stuck redex $g\ x\ t^n$, it "produces" a new dag where $g\ x\ t^n$ has been instantiated to $g\ (c\ x^m)\ t^n$ for each pattern $c\ x^m$ defined by g. The variables x^m need to be fresh to avoid them being captured by variables already existing in the dag. Each of these instantiated dags will allow further rewrites to take place, since each appropriate right-hand side of g now can be unfolded.

It is easy to see how I can create a transformation trace tree for a program q and an initial term t: First, pick some injection realisation and label the root of the transformation trace tree by a dag created from t. Then, repeatedly add new leaves to the transformation trace tree by using the relation $\xrightarrow[\mathsf{drive(q\leftarrow)}]{}$ to drive existing leaves.

Example 32 Consider again the well-known append program

$$\begin{aligned}
&\mathit{append}\ []\ \mathit{ys} &&= \mathit{ys}\\
&\mathit{append}\ (x : \mathit{xs})\ \mathit{ys} &&= x : (\mathit{append}\ \mathit{xs}\ \mathit{ys})\ ,
\end{aligned}$$

and the term '*append* (*append* *xs* *ys*) *zs*'. The transformation trace tree for this term is shown in Figure 9.1. The dag nodes that have solid addresses represent the frontier (i.e., they are marked). The transformation trace tree is finite because both leftmost leaves at the bottom are identical to an ancestor, when comparing the terms obtained by extracting at the marked addresses modulo renaming of variables. □

The use of dags poses a problem for extracting the transformed program from the transformation trace tree, since a dag can have several marked nodes, as seen from Example 32. How should such a frontier be interpreted? It seems natural to interpret the frontier as a *tuple* (or sequence) of terms. The program extraction thus has to deal with tuples of terms. To this end, I will allow the extraction process to emit programs which contain two extra syntactic constructs, namely tuple introduction $\langle \mathsf{t}^n \rangle$ and tuple elimination $\mathbf{let}\ \langle \mathsf{x}^n \rangle = \mathsf{t}'\ \mathbf{in}\ \mathsf{t}$. As you will see later, each tuple construct can either be completely eliminated from the extracted program, or it can be expressed directly in the non-extended language.

Example 33 By using tuple constructs, I can extract the following program from the transformation trace tree in Figures 9.1 and 9.2.

$$\begin{aligned}
&g_{\langle\rangle}\ \mathsf{NIL}\ \mathit{ys}\ \mathit{zs} &&= g_{\langle 11\rangle}\ \mathit{ys}\ \mathit{zs}\\
&g_{\langle\rangle}\ (\mathsf{CONS}\ w\ \mathit{ws})\ \mathit{ys}\ \mathit{zs} &&= \mathbf{let}\ \langle u\ ,\ v\rangle = \big\langle w\ ,\ (g_{\langle\rangle}\ \mathit{ws}\ \mathit{ys}\ \mathit{zs})\big\rangle\ \mathbf{in}\ \mathsf{CONS}\ u\ v\\
&g_{\langle 11\rangle}\ \mathsf{NIL}\ \mathit{zs} &&= \mathit{zs}\\
&g_{\langle 11\rangle}\ (\mathsf{CONS}\ w\ \mathit{ws})\ \mathit{zs} &&= \mathbf{let}\ \langle u\ ,\ v\rangle = \big\langle w\ ,\ (g_{\langle 11\rangle}\ \mathit{ws}\ \mathit{zs})\big\rangle\ \mathbf{in}\ \mathsf{CONS}\ u\ v
\end{aligned}$$

Eliminating the tuple constructs gives me the desired program:

$$\begin{aligned}
&g_{\langle\rangle}\ \mathsf{NIL}\ \mathit{ys}\ \mathit{zs} &&= g_{\langle 11\rangle}\ \mathit{ys}\ \mathit{zs}\\
&g_{\langle\rangle}\ (\mathsf{CONS}\ w\ \mathit{ws})\ \mathit{ys}\ \mathit{zs} &&= \mathsf{CONS}\ w\ (g_{\langle\rangle}\ \mathit{ws}\ \mathit{ys}\ \mathit{zs})\\
&g_{\langle 11\rangle}\ \mathsf{NIL}\ \mathit{zs} &&= \mathit{zs}\\
&g_{\langle 11\rangle}\ (\mathsf{CONS}\ w\ \mathit{ws})\ \mathit{zs} &&= \mathsf{CONS}\ w\ (g_{\langle 11\rangle}\ \mathit{ws}\ \mathit{zs})\ .
\end{aligned}$$

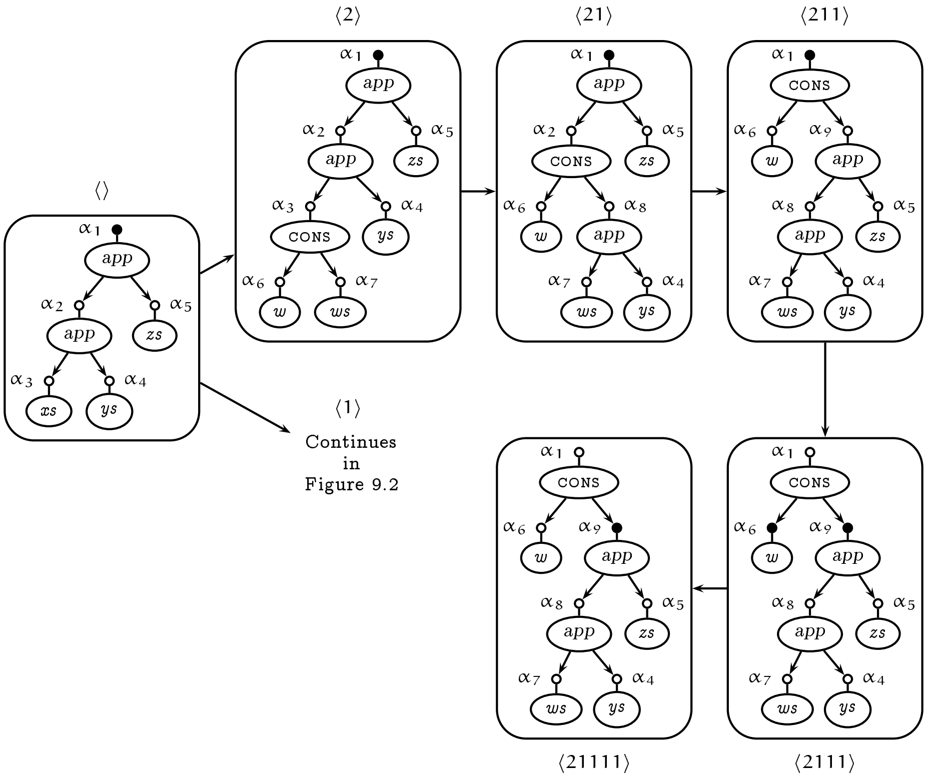


Figure 9.1: Transformation of double append using dags

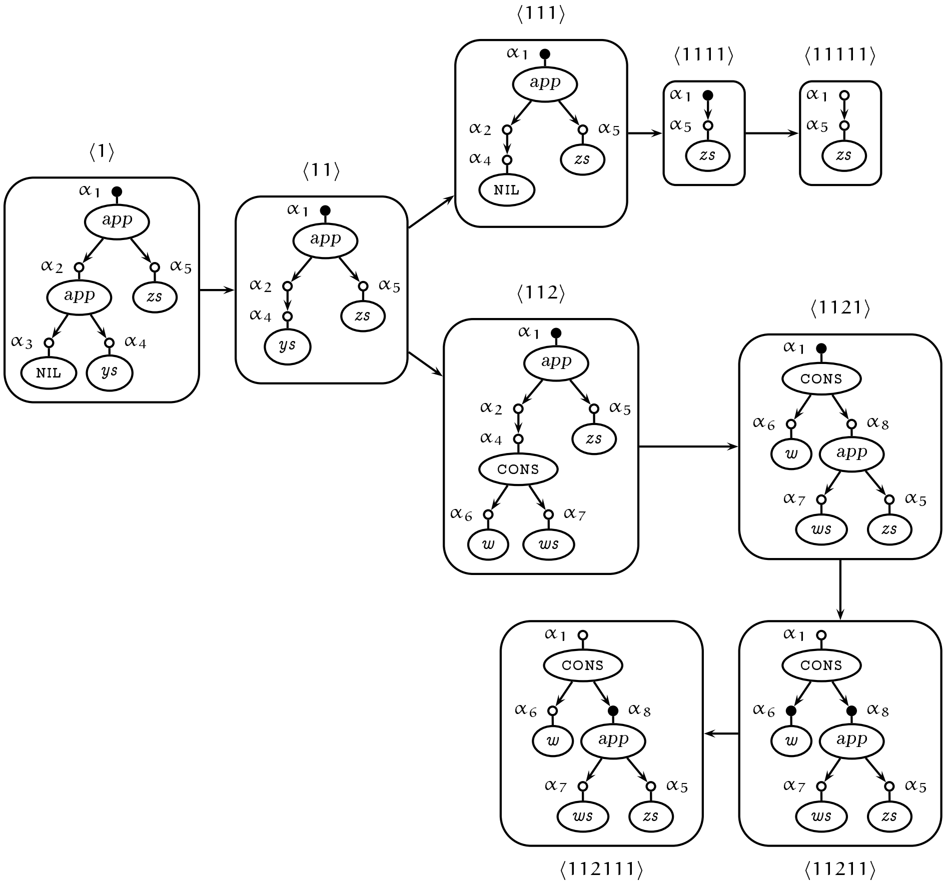


Figure 9.2: Transformation of double append using dags (cont.)

□

9.2 Relation to term-based driving

The last example should hopefully have illustrated that the dag-based program transformation technique can achieve the same effects as deforestation. This property can be stated more precisely.

Lemma 30 *Let* $\leftarrow$ *be the duplicating injection. For any term* t*, if*

$$\{t' \mid t \xrightarrow[\mathsf{deforest}(q)]{} t'\} = \{t^n\} \ ,$$

then either

$$\left\{ \langle\Gamma\rangle_\alpha \,\middle|\, \mathsf{tree}(t) \xrightarrow[\mathsf{drive}(q \leftarrow)]{+} \Gamma^{\langle\alpha\rangle} \right\} = \{t^n\} \ ,$$

or

$$\mathsf{tree}(t) \xrightarrow[\mathsf{drive}(q \leftarrow)]{} \Gamma^{\langle\alpha^n\rangle} \wedge \langle\Gamma\rangle_{\alpha_i} = t_i$$

for all $i \in \{1 \ldots n\}$.

PROOF (sketch). By case analysis on the structure of t. If $t = e[r]$ is unfolded by use of rule FCALL or rule MATCH, then it is easy to verify that the corresponding unfolding is performed by a $\xrightarrow[\mathsf{drive}(q \leftarrow)]{}$ step. Otherwise, if $t = e[g\ x\ t^n] \xrightarrow[\mathsf{deforest}(q)]{} t'$ by speculatively executing t (by use of rule SPEC), then, because every occurrence of variable x is a distinct node in $\mathsf{tree}(t)$, it is not hard to show that

$$\mathsf{tree}(t) \xrightarrow[\mathsf{drive}(q \leftarrow)]{} \Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{drive}(q \leftarrow)]{} \Gamma^{\langle\alpha\rangle}$$

such that $\langle\Delta\rangle_\alpha = e[g\ (c\ x^m)\ t^n]$ for some $c\ x^m$, and then it is easy to see that $\langle\Gamma\rangle_\alpha = t'$. Otherwise, if $t = c\ t^n \xrightarrow[\mathsf{deforest}(q)]{} t_i$ for all $i \in \{1 \ldots n\}$ by use of rule INDEP, then it can easily be seen that

$$\mathsf{tree}(t) \xrightarrow[\mathsf{drive}(q \leftarrow)]{} \Gamma^{\langle\alpha^n\rangle}$$

such that $\langle\Gamma\rangle_{\alpha_i} = t_i$ for all $i \in \{1 \ldots n\}$. □

A similar statement can be made about the relation between positive supercompilation and dag-based driving. More specifically, if $\mathsf{stddag}(t) = \Delta^{\langle\alpha\rangle}$ and

$$t \xrightarrow[\mathsf{posscp}(q)]{*} t' \ ,$$

then

$$\Delta^{\langle\alpha\rangle} \xrightarrow[\mathsf{drive}(q\leftarrow)]{*} \Gamma^{\langle\alpha^n\rangle} \wedge \langle\Gamma\rangle_\alpha = t'$$

when using the duplicating injection modified so that it does not duplicate variables.

The difference between deforestation and positive supercompilation then becomes strikingly clear: It is merely a matter of term representation. More surprisingly, when using the *collapsing* injection during driving, some of the effects seen in tupling can be achieved. I will present an example of such effects in Section 9.7, after I have presented the complete algorithm for dag-based program transformation.

9.3 Abstract program transformers

As seen in previous chapters, creating a transformation trace tree by driving alone (in the previously defined manner) hardly ever terminates, that is, the transformation trace tree will grow unboundedly. Intuitively, I therefore need to monitor the construction of the transformation trace tree in such a way that I am able to intervene when there is a danger of non-termination. In the next two sections I will deal with the problem of detecting possible non-termination of driving, and in Section 9.5 I will show what to do when an intervention is needed. In Section 9.6 I will present the complete algorithm for dag-based supercompilation, which will incorporate the monitoring and generalisations into the basic driving mechanism.

The fundamental tools used in ensuring termination are *well-founded quasi-orders* and *well-quasi-orders*.

Definition 59 (wfqo,wqo)

1. Let S be a set with a relation $\succeq \subseteq S \times S$. If $\succeq$ is transitive and reflexive, then $\langle S , \succeq\rangle$ is a *quasi-order*.[1] I will write $s \succ s'$ iff $s \succeq s'$ and $s' \not\succeq s$.

2. A quasi-order $\langle S , \succeq\rangle$ is *well-founded* if there is no infinite sequence $\langle s_0 , s_1 , \ldots\rangle \in S^\omega$ with $s_0 \succ s_1 \succ \ldots$.

[1] Quasi-orders also go by the name *pre-orders*.

3. A quasi-order $\langle S , \succeq \rangle$ is a *well-quasi-order* if, for any infinite sequence $\langle s_0 , s_1 , \ldots \rangle \in S^\omega$, there are i and $j \in \mathbb{N}$ such that $j > i$ and $s_j \succeq s_i$. □

The usefulness of wqo's is due to Kruskal's Tree Theorem as described in Section 5.2 (Theorem 2): If the nodes of the tree are well-quasi-ordered, any infinite transformation trace tree will have an infinite branch where some node is less than or equal to some successor node. Hence, when such a situation occurs during construction of the transformation trace tree, an intervention might be needed. Similarly for well-founded quasi-orders; as long as nodes in the transformation trace tree are strictly decreasing, it is not possible to construct an infinite branch.

The nice properties of wqo's and wfqo's are exploited in the framework of *abstract program transformers*, a theoretical framework described by Sørensen [108]. In this framework, he states some simple, sufficient conditions that will ensure the termination of a program transformer. I will here repeat some these conditions in order to explain why the algorithm for dag-based supercompilation is formulated the way it is.

Definition 60 Let S be a set of tree labels. An *abstract program transformer* is a map $M \in \mathit{Tree}\ S \Rightarrow \mathit{Tree}\ S$. □

An abstract program transformer is thus a function that, given a *process* tree, returns a new process tree. The process tree is what I have so-far called the transformation trace tree, and the labels of the process tree is thus the set of marked dags. The idea of the above formulation is that the abstract program transformer performs a relatively simple transformation step (e.g., unfolding a redex) each time it is invoked, and construction of a full process tree is achieved by iterating the transformer M over an initial process tree τ. The iteration stops when $M\tau = \tau$.

Definition 61 Let $M \in \mathit{Tree}\ S \Rightarrow \mathit{Tree}\ S$ be an abstract program transformer and $\tau \in \mathit{Tree}\ S$.

1. $M^0\tau \overset{\text{def}}{=} \tau$ and $M^{i+1}\tau \overset{\text{def}}{=} M^i(M\tau)$.

2. M *terminates on* τ iff $M^i\tau = M^{i+1}\tau$ for some $i \in \mathbb{N}$.

3. M *terminates* iff M terminates on all singleton trees. □

The idea for ensuring termination of an abstract program transformer M is the following. Firstly, iterative application of M should produce a *Cauchy* sequence of process trees, in the sense that, for any depth n, the process trees should be identical down to depth n from some point in the sequence. An M which possesses this quality is called a Cauchy transformer. A particular class of Cauchy transformer is the transformers that operates by either adding new nodes to the process tree, or replaces a whole sub-tree by a another sub-tree for which the root label is strictly smaller.

Definition 62 (Proposition 38 in [108]) Let $\langle S\,, \succeq\rangle$ be a well-founded quasi-order and $M \in \mathit{Tree}\, S \Rightarrow \mathit{Tree}\, S$. M is *Cauchy* iff, for all $\tau \in \mathit{Tree}\, S$, it holds that $M\tau = \tau[\eta := \tau']$ for some $\eta \in \mathbb{N}_1^{\star}$ and $\tau' \in \mathit{Tree}\, S$ where

1. $\eta \in \mathrm{leaves}_\tau \wedge \tau\eta = \tau'\langle\rangle$, or

2. $\tau\eta \succ \tau'\langle\rangle$. □

Remark 10 The term Cauchy has a well-defined meaning in [108], but here I have simply used it to denote a special class of transformers.

Secondly, M should maintain an invariant on the process trees that ensures that the process trees do not grow unboundedly. More specifically, M should maintain a predicate that is false for all *infinite* trees.

Definition 63 An abstract program transformer M *maintains a predicate* $p \in \mathit{Tree}^{\omega}\, S \Rightarrow \mathbb{B}$ iff, for all singleton $\tau \in \mathit{Tree}\, S$ and $i \in \mathbb{N}$, it holds that $p(M^i\tau) = \mathbf{true}$. □

In lieu of the nice properties of well-quasi-orders and well-founded quasi-orders w.r.t. infinite trees, it is reasonable to construct a predicate based on either wqos or wfqos. In practice, it is necessary to relax the ordering on the nodes in the process tree by combining several wqos and wfqos. The following generic definition can be used to construct appropriate predicates by partitioning the nodes of a process tree into layers, each equipped with its own ordering. When two nodes from the same layer are compared, intermediate nodes from lower layers are ignored. You can see an example of layers in Figure 9.3.

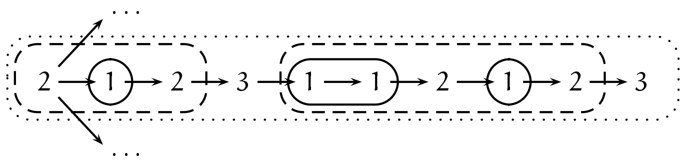


Figure 9.3: Layers in a tree. Nodes are compared to ancestors of the same class as long as there is no intervening node of a higher class.

Definition 64 Let $\tau \in \mathit{Tree}\ S$ be a tree, let $\{A^n\} = \mathscr{D}\tau$ be a partitioning of the nodes of τ, and let $\{\geq^n\}$ be well-quasi-orders or well-founded quasi-orders on $S \times S$. Then the predicate $p \in \mathit{Tree}^{\omega}\ S \Rightarrow \mathbb{B}$ defined by

$$p\ \tau \overset{\text{def}}{=} \begin{cases} \textbf{false}, & \text{if } \exists i \in \{1 \ldots n\} \\ & \left(\exists\{\eta\,,\,\mu\} \subseteq A_i \left(\begin{array}{c} \mu \notin \text{leaves}_\tau \wedge \eta <_\tau \mu \\ \wedge \\ \forall \nu \left(\eta <_\tau \nu <_\tau \mu \Rightarrow \nu \in \bigcup_{j=1}^{i} A_j \right) \\ \wedge \\ \begin{cases} \tau\eta \leq_i \tau\mu, & \text{if } \langle A_i\,,\,\geq_i \rangle \text{ wqo} \\ \tau\eta \not\geq_i \tau\mu, & \text{if } \langle A_i\,,\,\geq_i \rangle \text{ wfqo} \end{cases} \end{array} \right) \right) \\ \textbf{true}, & \text{otherwise.} \end{cases}$$

is *finitary and continuous.* □

Remark 11 The terms finitary and continuous have a well-defined meaning, but here I have simply used them to denote a special class of predicates. That the previously defined predicate is indeed finitary and continuous follows easily by extending Proposition 41 in [108] by thinning out each layer, after which Proposition 46 in [108] can be applied.

A sufficient condition for ensuring termination of an abstract program transformer can now be formulated.

Lemma 31 (Sørensen [108], Theorem 33) *Let* $M \in \mathit{Tree}\ S \Rightarrow \mathit{Tree}\ S$ *maintain predicate* $p \in \mathit{Tree}\ S \Rightarrow \mathbb{B}$*. If* M *is Cauchy and* p *is finitary and continuous, then* M *terminates.*

In the next section I will construct a suitable well-quasi-order on dags, and in Section 9.5 I will define the various generalisation operations that can be invoked when termination is endangered. Finally, in Section 9.6, I will define an algorithm for dag-based supercompilation and employ the preceding theory to prove that this algorithm terminates.

9.4 A well-quasi-order on dags

I will base the generalisation operations on the homeomorphic-embedding relation, as in the previous chapters. The following definition will be used to infer *cut points* in a dag Δ w.r.t. an embedded dag Γ; and explanation follows the definition.

Definition 65 I will say that $\Delta^{\langle\alpha\rangle}$ *is embedded in* $\Gamma^{\langle\beta\rangle}$ *by cut points* A, if

$$\Delta^{\langle\alpha\rangle} \trianglelefteq_{\text{hom}} \Gamma^{\langle\beta\rangle} \text{ by } A$$

follows from the inference system in Figure 9.4. I write $\Delta^{\langle\alpha\rangle} \not\trianglelefteq_{\text{hom}} \Gamma^{\langle\beta\rangle}$ as a shorthand for $\nexists A \left(\Delta^{\langle\alpha\rangle} \trianglelefteq_{\text{hom}} \Gamma^{\langle\beta\rangle} \text{ by } A \right)$. □

Remark 12 Despite of the premise $(\Delta^{\langle\alpha\rangle} \not\trianglelefteq_{\text{hom}} \Gamma^{\langle\beta\cdot\rangle})^n$ in rule COUPLE, the inference system in Figure 9.4 is constructive because the dags are finite and non-cyclic.

The INDIRECT rule applies whenever $\Delta^{\langle\alpha\rangle}$ is an indirection, in which case the inferred embeddings, if any, are the same those inferred for the indirection. Similarly, $\Delta^{\langle\alpha\rangle}$ is not an indirection but $\Gamma^{\langle\beta\rangle}$ is, rule INDIRECT' says that the inferred embeddings, if any, are the same those inferred for the indirection.

The last three rules (VAR, DIVE and COUPLE) can be explained by contrasting them to the homeomorphic embedding on terms (cf. Definition 34):

$$\frac{}{x \trianglelefteq_{\text{hom}} y} \qquad \frac{\exists i \in \{1,\dots,n\}\ (\, t \trianglelefteq_{\text{hom}} t_i \,)}{t \trianglelefteq_{\text{hom}} s\ t^n} \qquad \frac{\forall i \in \{1,\dots,n\}\ (\, t_i \trianglelefteq_{\text{hom}} t_i' \,)}{s\ t^n \trianglelefteq_{\text{hom}} s\ t'^n}\ . \tag{9.1}$$

$$\Delta \in \mathit{Dag}, \Gamma \in \mathit{Dag}, \alpha \in \mathit{Addr}, \beta \in \mathit{Addr}, \gamma \in \mathit{Addr}, A \subseteq \mathit{Addr}$$
$$s \in \mathit{Function} \cup \mathit{Matcher} \cup \mathit{Constructor}, n \geq 0$$

$$\textsc{indirect}\quad \frac{\Delta\alpha = \gamma \qquad \Delta^{\langle\gamma\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle} \text{ by } A}{\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle} \text{ by } A}$$

$$\textsc{indirect'}\quad \frac{\Delta \vdash \alpha \rightsquigarrow \alpha \qquad \Delta\beta = \gamma \qquad \Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\gamma\rangle} \text{ by } A}{\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle} \text{ by } A}$$

$$\textsc{var}\quad \frac{\Delta\alpha \in \mathit{Variable} \qquad \Gamma\beta \in \mathit{Variable}}{\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle} \text{ by } \{\beta\}}$$

$$\textsc{dive}\quad \frac{\begin{array}{c} \Delta \vdash \alpha \rightsquigarrow \alpha \qquad\qquad \Gamma\beta = s\ \beta^n \\ A = \bigcup\left\{ B \,\middle|\, \Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\gamma\rangle} \text{ by } B \wedge \gamma \in \{\beta^n\}\right\} \qquad A \neq \varnothing \end{array}}{\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle} \text{ by } A}$$

$$\textsc{couple}\quad \frac{\begin{array}{cc} \Delta\alpha = s\ \alpha^n & \Gamma\beta = s\ \beta^n \\ (\Delta^{\langle\alpha\rangle} \not\trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta.\rangle})^n & (\Delta^{\langle\alpha.\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta.\rangle} \text{ by } A.)^n \end{array}}{\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle} \text{ by } \{\beta\}}$$

Figure 9.4: Dag embeddings

$\{ x , y , v , w \} \subseteq \mathit{Variable}, \{ f , g \} \subseteq \mathit{Symbol}$

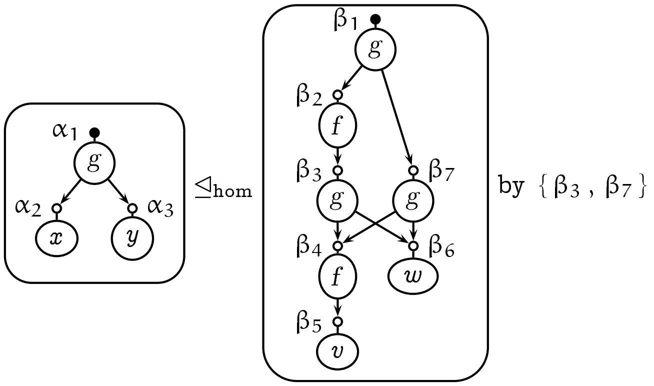


Figure 9.5: Dag embedding and cut points

The VAR rule applies when both $\Delta^{\langle\alpha\rangle}$ and $\Gamma^{\langle\beta\rangle}$ represent variables, just as the leftmost rule above, but the VAR rule provides the additionally information that $\Delta^{\langle\alpha\rangle}$ is embedded in $\Gamma^{\langle\beta\rangle}$ at address β.

The DIVE rule applies when $\Delta^{\langle\alpha\rangle}$ is embedded in at least one of the children of $\Gamma^{\langle\beta\rangle}$, just as the rule in the middle of Equation 9.1, but the DIVE rule provides additional information about *where* $\Delta^{\langle\alpha\rangle}$ is embedded in the children.

The COUPLE rule applies when $\Delta^{\langle\alpha\rangle}$ and $\Gamma^{\langle\beta\rangle}$ have the same non-variable root symbol and all their children are pairwise embedded. If that is the case, the additional information that $\Delta^{\langle\alpha\rangle}$ is embedded in $\Gamma^{\langle\beta\rangle}$ at address β can be inferred. But in contrast to the right-hand rule in Equation 9.1, the COUPLE rule requires that $\Delta^{\langle\alpha\rangle}$ is *not* embedded in any of the children of $\Gamma^{\langle\beta\rangle}$.

The DIVE rule thus has priority over the COUPLE rule, meaning that, if there is a choice in the selection of two different cut points, the one that succeeds the other is chosen (i.e., the one furthest from the root). Both this priority and the accumulation of cut points are illustrated in the following example.

Example 34 Using the convention that marked addresses are solid, whereas unmarked addresses are hollow, the result depicted in Figure 9.5 is inferable from the inference system in Figure 9.4. The example illustrates that the dag-embedding relation identifies all the smallest sub-dags in which the embedding occurs. □

The dag-embedding relation ensures that the identified cut points are unique.

Lemma 32 $\Delta^{\langle\alpha\rangle} \trianglelefteq_{hom} \Gamma^{\langle\beta\rangle}$ *by* A *is deterministic, in the sense that*

$$\Delta^{\langle\alpha\rangle} \trianglelefteq_{hom} \Gamma^{\langle\beta\rangle} \text{ by } A \wedge \Delta^{\langle\alpha\rangle} \trianglelefteq_{hom} \Gamma^{\langle\beta\rangle} \text{ by } B \Rightarrow A = B \ .$$

PROOF. I assume $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by A and $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by B and proceed by induction on the height of the inference tree. If $\Delta\alpha$ is an indirection, only rule INDIRECT applies, and thus by the induction hypothesis $A = B$. If $\Delta\alpha$ is not an indirection but $\Gamma\beta$ is an indirection, only rule INDIRECT' applies, and thus by the induction hypothesis $A = B$. If neither $\Delta\alpha$ nor $\Gamma\beta$ are indirections, then, if both $\Delta\alpha \in \mathit{Variable}$ and $\Gamma\beta \in \mathit{Variable}$, then only rule VAR applies, which trivially makes $A = B$. Otherwise, $\Delta\beta = s\ \beta^n$. If $n = 0$, then only rule COUPLE applies, so trivially $A = B$. If $n > 0$, then either the set $A = \bigcup \left\{ B \;\middle|\; \Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\gamma\rangle} \text{ by } B \wedge \gamma \in \{\beta^n\} \right\}$ is empty, in which case only rule COUPLE applies, and thus trivially $A = B$; or A is non-empty, in which case only rule DIVE applies, and, by the induction hypothesis, each of the sets comprising the union are unique, so also A is unique. □

The dag-embedding relation corresponds to the term-embedding relation.

Lemma 33 $\exists A \neq \varnothing \left(\Delta^{\langle\alpha\rangle} \trianglelefteq_{hom} \Gamma^{\langle\beta\rangle} \text{ by } A \right) \Leftrightarrow \langle\Delta\rangle_\alpha \trianglelefteq_{hom} \langle\Gamma\rangle_\beta$.

PROOF. ($\Rightarrow$) I assume $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by A (for some A) and proceed by induction on the height of the inference tree, analysing each case in Figure 9.4.

If rule INDIRECT applies, then $\Delta\alpha = \gamma$ and, by the induction hypothesis, $\langle\Delta\rangle_\gamma \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_\beta$. Since $\langle\Delta\rangle_\gamma = \langle\Delta\rangle_\alpha$, also $\langle\Delta\rangle_\alpha \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_\beta$, as required. Rule INDIRECT' is similar.

If rule VAR applies, trivially $\langle\Delta\rangle_\alpha \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_\beta$.

If rule DIVE applies, then $\Gamma\beta = s\ \beta^n$ and $\exists B \left(\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta_i\rangle} \text{ by } B \right)$ (for some i). By the induction hypothesis $\langle\Delta\rangle_\alpha \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_{\beta_i}$, and thus also $\langle\Delta\rangle_\alpha \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_\beta$.

If rule COUPLE applies, then $\Delta\alpha = s\ \alpha^n$, $\Gamma\beta = s\ \beta^n$ and $(\Delta^{\langle\alpha.\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta.\rangle} \text{ by } B.)^n$. By the induction hypothesis $(\langle\Delta\rangle_{\alpha.} \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_{\beta.})^n$, and thus also $\langle\Delta\rangle_\alpha \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_\beta$.

($\Leftarrow$) I assume $\langle\Delta\rangle_\alpha \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_\beta$ and proceed by structural induction on one of the inference trees that witnesses $\langle\Delta\rangle_\alpha \trianglelefteq_{\mathrm{hom}} \langle\Gamma\rangle_\beta$.

If both $\langle\!\langle\Delta\rangle\!\rangle_\alpha = x \in \mathit{Variable}$ and $\langle\!\langle\Gamma\rangle\!\rangle_\beta = y \in \mathit{Variable}$, then $\Delta \vdash \alpha \rightsquigarrow \alpha'$ for some α' such that $\Delta\alpha' \in \mathit{Variable}$, and $\Gamma \vdash \beta \rightsquigarrow \beta'$ for some β' such that $\Delta\beta' \in \mathit{Variable}$. By rule VAR, $\Delta^{\langle\alpha'\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta'\rangle}$ by $\{\,\beta'\,\}$. By a finite number of applications of rule INDIRECT’, $\Delta^{\langle\alpha'\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by $\{\,\beta'\,\}$, and by a finite number of applications of rule INDIRECT, $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by $\{\,\beta'\,\}$, as required.

If $\langle\!\langle\Gamma\rangle\!\rangle_\beta = s\ t^n$, then $\Gamma \vdash \beta \rightsquigarrow \beta'$ for some β' such that $\Delta\beta' = s\ \beta^n$, and $\Delta \vdash \alpha \rightsquigarrow \alpha'$ for some α'. If $n = 0$, then $\Delta\alpha' = s$, and thus $\Delta^{\langle\alpha'\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta'\rangle}$ by $\{\,\beta'\,\}$ by rule COUPLE. As in the case above, by a finite number of applications of rules INDIRECT and INDIRECT’, $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by $\{\,\beta'\,\}$, as required.

If $n > 0$, then if there is a witness for $\langle\!\langle\Delta\rangle\!\rangle_{\alpha'} \trianglelefteq_{\mathrm{hom}} \langle\!\langle\Gamma\rangle\!\rangle_{\beta_i}$ for some $i \in \{\,1 \ldots n\,\}$, then, by the induction hypothesis, $\Delta^{\langle\alpha'\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta_i\rangle}$ by B such that $B \neq \varnothing$, and by rule DIVE, $\Delta^{\langle\alpha'\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta'\rangle}$ by B. Again, by a finite number of applications of rules INDIRECT and INDIRECT’, $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by B, as required.

Otherwise, there is no $B \neq \varnothing$ such that $\Delta^{\langle\alpha'\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta_i\rangle}$ by B for any $i \in \{\,1 \ldots n\,\}$, since that would imply $\langle\!\langle\Delta\rangle\!\rangle_{\alpha'} \trianglelefteq_{\mathrm{hom}} \langle\!\langle\Gamma\rangle\!\rangle_{\beta_i}$. Moreover, there must be a witness for $\langle\!\langle\Delta\rangle\!\rangle_{\alpha'} \trianglelefteq_{\mathrm{hom}} \langle\!\langle\Gamma\rangle\!\rangle_{\beta'}$ and thus $\Delta\alpha' = s\ \alpha^n$ and $(\langle\!\langle\Delta\rangle\!\rangle_{\alpha.} \trianglelefteq_{\mathrm{hom}} \langle\!\langle\Gamma\rangle\!\rangle_{\beta.})^n$. By the induction hypothesis, $(\Delta^{\langle\alpha.\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta.\rangle}$ by $B.)^n$, and thus $\Delta^{\langle\alpha'\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta'\rangle}$ by $\{\,\beta'\,\}$ by rule COUPLE. Again, by a finite number of applications of rules INDIRECT and INDIRECT’, $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by $\{\,\beta'\,\}$, as required. □

The cut-points identify the precise location of trouble spots in lager dag, in the sense that they themselves embed the smaller dag.

Lemma 34 $\Delta^{\langle\alpha\rangle} \trianglelefteq_{hom} \Gamma^{\langle\beta_0\rangle}$ *by* $\{\,\beta^n\,\} \Rightarrow (\langle\!\langle\Delta\rangle\!\rangle_\alpha \trianglelefteq_{hom} \langle\!\langle\Gamma\rangle\!\rangle_{\beta.})^n$.

PROOF. Follows easily by extending the proof of the right-to-left direction of Lemma 33. □

Lemma 35 *Deciding whether* $\Delta^{\langle\alpha\rangle} \trianglelefteq_{hom} \Gamma^{\langle\beta\rangle}$ *by* B *can be done in*

$$O\,(|\{\,\gamma \mid \Delta \vdash \alpha \overset{*}{\multimap} \gamma\,\}| \cdot |\{\,\gamma \mid \Gamma \vdash \beta \overset{*}{\multimap} \gamma\,\}|)$$

time and space.

PROOF. Let $A = \{\,\gamma \mid \Delta \vdash \alpha \overset{*}{\multimap} \gamma\,\}$ and $B = \{\,\gamma \mid \Gamma \vdash \beta \overset{*}{\multimap} \gamma\,\}$. Construct an $A \times B$ table where each element is initialised to "no answer" (na). Other possible entries are NO (i.e., $\Delta^{\langle\alpha\rangle} \not\trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$), β (i.e., $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by $\{\,\beta\,\}$), and $\langle\beta^n\rangle$

(which means $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\text{hom}} \Gamma^{\langle\beta\rangle}$ by B where B should be collected recursively from the entries of table$\langle\alpha\,,\,\beta_1\rangle$ to table$\langle\alpha\,,\,\beta_n\rangle$). Deciding $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\text{hom}} \Gamma^{\langle\beta\rangle}$ by B can be done recursively as shown in Figure 9.6. The algorithm in Figure 9.6 is clearly a decision procedure for the inference system in Figure 9.4. Each entry calculation in the table can be done in constant time, since the number of children is bounded by the maximal arity of symbols. Each entry is calculated only once, and since the table is of size $A \times B$, the time and space used is $O(A \cdot B)$. □

Example 35 Consider again the dags shown in Figure 9.5. The table constructed to calculate the cut points is

	α_1	α_2	α_3
β_1	$\langle\beta_2\,,\,\beta_7\rangle$	na	na
β_2	$\langle\beta_3\rangle$	na	na
β_3	β_3	na	na
β_4	NO	$\langle\beta_5\rangle$	na
β_5	NO	β_5	β_5
β_6	NO	na	β_6
β_7	β_7	na	na

The result can be extracted by starting from the $\langle\alpha_1\,,\,\beta_1\rangle$ entry, which says that the result is the union of the $\langle\alpha_1\,,\,\beta_2\rangle$ and $\langle\alpha_1\,,\,\beta_7\rangle$ entries. The $\langle\alpha_1\,,\,\beta_2\rangle$ entry says that its result is found in the the $\langle\alpha_1\,,\,\beta_3\rangle$ entry, which is $\{\,\beta_3\,\}$. The $\langle\alpha_1\,,\,\beta_7\rangle$ entry is $\{\,\beta_7\,\}$. Thus the final result is $\{\,\beta_3\,,\,\beta_7\,\}$, as expected. □

The reason for identifying cut points is that they subsequently can be used to carve out all embeddings. I will return to the issues of *how* to perform the generalisation in the next section.

The preceding shows that the concept of homeomorphic embedding on terms can be smoothly transferred to single-marked dags. Indeed, the concept is easily extended to multi-marked dags, as long as the dags have the same number of marks.

Definition 66 If $\Delta^{\langle\alpha^n\rangle}$ and $\Gamma^{\langle\beta^n\rangle} \in \mathit{Mdag}$, then

$$\Delta^{\langle\alpha^n\rangle} \trianglelefteq_{\text{hom}} \Gamma^{\langle\beta^n\rangle} \text{ by } A \Leftrightarrow \exists A^n \left(\bigwedge (\Delta^{\langle\alpha_\cdot\rangle} \trianglelefteq_{\text{hom}} \Gamma^{\langle\beta_\cdot\rangle} \text{ by } A_\cdot)^n \wedge A = \bigcup A^n \right) .$$

$\{\Delta, \Gamma\} \subseteq \mathit{Dag}, \{\alpha, \beta, \gamma\} \subseteq \mathit{Addr}, \mathsf{s} \in \mathit{Symbol},$
table entries $= (\mathrm{na} + \mathrm{NO} + \mathit{Addr} + \mathit{Addr}^{\star})$

If table$\langle\alpha, \beta\rangle = \mathrm{na}$, then

1. If $\Delta\alpha \in \mathit{Variable}$ and $\Gamma\beta \in \mathit{Variable}$, set table$\langle\alpha, \beta\rangle := \beta$. (VAR)
2. Otherwise, if $\Delta\alpha = \gamma$, calculate the entry for table$\langle\gamma, \beta\rangle$, and set table$\langle\alpha, \beta\rangle :=$ table$\langle\gamma, \beta\rangle$. (INDIRECT)
3. Otherwise, if $\Delta \vdash \alpha \rightsquigarrow \alpha$ and $\Gamma\beta = \gamma$, calculate the entry for table$\langle\alpha, \gamma\rangle$, and set table$\langle\alpha, \beta\rangle :=$ table$\langle\alpha, \gamma\rangle$. (INDIRECT')
4. Otherwise, if $\Delta \vdash \alpha \rightsquigarrow \alpha$ and $\Gamma\beta = \mathsf{s}\ \beta^n$, calculate the entries for table$\langle\alpha, \beta_1\rangle$ to table$\langle\alpha, \beta_n\rangle$.
   (a) If any of these entries are different from NO, then set table$\langle\alpha, \beta\rangle := \langle\beta^n\rangle$. (DIVE)
   (b) Otherwise, if $\Delta\alpha = \mathsf{s}\ \alpha^n$, calculate the entries for table$\langle\alpha_1, \beta_1\rangle$ to table$\langle\alpha_n, \beta_n\rangle$. If all these entries are different from NO, then set table$\langle\alpha, \beta\rangle := \beta$. (COUPLE)
   (c) Otherwise, set table$\langle\alpha, \beta\rangle :=$ NO.
5. Otherwise, set table$\langle\alpha, \beta\rangle :=$ NO.

If table$\langle\alpha, \beta\rangle =$ NO, then $\Delta^{\langle\alpha\rangle} \ntrianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$.

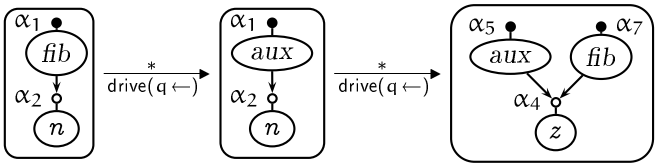
If table$\langle\alpha, \beta\rangle = \gamma$, then $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by $\{\gamma\}$.

If table$\langle\alpha, \beta\rangle = \langle\beta^n\rangle$, then recursively traverse table$\langle\alpha, \beta_1\rangle$ to table$\langle\alpha, \beta_n\rangle$, collecting all γ for which table$\langle\alpha, \gamma\rangle = \gamma$ (i.e., $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\gamma\rangle}$ by $\{\gamma\}$), and call this set B. The final result is $\Delta^{\langle\alpha\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta\rangle}$ by B.

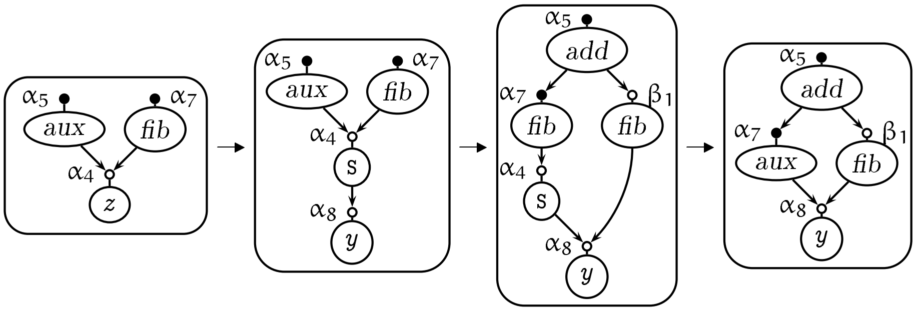
Figure 9.6: Dag embedding algorithm with memoisation

The above definition of embedding is *not* a well-quasi-order on dags, since an infinite sequence of dags can be constructed by allowing the number of marks to grow unboundedly. I therefore need to devise a relation that will relate enough dags with different number of marks. A naïve relation would be the following: $\Delta^{\langle\alpha^n\rangle} \trianglelefteq_{\text{hom}} \Gamma^{\langle\beta^m\rangle}$ by A iff there is an injection $1 \leq j_1 < j_2 < \cdots < j_n \leq m$ such that $\Delta^{\langle\alpha_i\rangle} \trianglelefteq_{\text{hom}} \Gamma^{\left\langle\beta_{j_i}\right\rangle}$ by A_i for all $i \in \{1 \ldots n\}$. However, this naïve extension is to strict, in the sense that it does not take *potential sharing of computation* into consideration.

Example 36 Let q be the Fibonacci program from Example 31 and $\leftarrow$ be the collapsing injection (Figure 8.5). Supercompiling the term '*fib n*' might lead to the following branch in the process tree:



The naïve extension suggested above tells me to stop driving the rightmost dag since it embeds a predecessor (two, indeed). But if I stop driving and break up the the rightmost dag, I would not exploit the sharing of computation that reveals itself after a few driving steps:



The sharing of computation stems from the instantiation of the variable z. By instantiating this variable, the seemingly unrelated functions *aux* and *fib* can both be unfolded and sharing of computation is discovered. □

The example above shows that it is beneficial to take sharing of variables into consideration when deciding whether or not to generalise a dag. I will therefore define a simple measurement that expresses the proportion between distinct variables and distinct function symbols.

Definition 67 The *variable ratio* function varratio $\in$ *Mdag* ⇨ $\mathbb{R}$ is defined by

$$\mathrm{varratio}\ \Delta^{A} \overset{\mathrm{def}}{=} \frac{\left|\mathrm{variables}\ \Delta^{A}\right|}{\left|\mathrm{functions}\ \Delta^{A}\right|}$$

where

$$\begin{array}{l}
\mathrm{reachable}\ \Delta^{A} \overset{\mathrm{def}}{=} \left\{\, \beta \;\middle|\; \alpha \in A \wedge \Delta \vdash \alpha \overset{*}{\multimap} \beta \,\right\} \\
\mathrm{variables}\ \Delta^{A} \overset{\mathrm{def}}{=} \left\{\, x \in \mathit{Variable} \;\middle|\; \beta \in \mathrm{reachable}\ \Delta^{A} \wedge \Delta\beta = x \,\right\} \\
\mathrm{functions}\ \Delta^{A} \overset{\mathrm{def}}{=} \left\{\, h \in \mathit{Function} \cup \mathit{Matcher} \;\middle|\; \beta \in \mathrm{reachable}\ \Delta^{A} \wedge \Delta\beta = h\ \alpha^{n} \,\right\}\ .
\end{array}$$

□

By using the variable ratio and the homeomorphic embedding relation, I can now construct a suitable well-quasi-order.

Definition 68 I will say that A *covers* $\Gamma^{\langle\beta^{m}\rangle}$ *w.r.t.* $\Delta^{\langle\alpha^{n}\rangle}$, denoted

$$\Delta^{\langle\alpha^{n}\rangle} \preceq \Gamma^{\langle\beta^{m}\rangle}\ \text{by}\ A$$

if, and only, if

$$\exists j^{n} \in \mathbb{N} \left(\begin{array}{c}
1 \leq j_1 < j_2 < \cdots < j_n \leq m \\
\wedge \\
\exists A^{n}\ \left(A = \bigcup A^{n} \wedge \left(\Delta^{\langle\alpha_{.}\rangle} \trianglelefteq_{\mathrm{hom}} \Gamma^{\langle\beta_{j_{.}}\rangle}\ \text{by}\ A_{.}\right)^{n} \right) \\
\wedge \\
n < m \Rightarrow \mathrm{varratio}\ \Delta^{\langle\alpha^{n}\rangle} \leq \mathrm{varratio}\ \Gamma^{\langle\beta^{m}\rangle}
\end{array} \right) \quad .$$

I will write $\Delta^{\langle\alpha^{n}\rangle} \preceq \Gamma^{\langle\beta^{m}\rangle}$ when $\exists A\ \left(\Delta^{\langle\alpha^{n}\rangle} \preceq \Gamma^{\langle\beta^{m}\rangle}\ \text{by}\ A\right)$. □

Lemma 36 *$\langle Mdag, \preceq\rangle$ is a well-quasi-order.*

PROOF. Assume $\Delta_0^{A_0}$, $\Delta_1^{A_1}$, $\Delta_1^{A_1}$, ... is an infinite sequence of marked dags. Since dag embedding is equivalent to term embedding (Lemma 33), $\trianglerighteq$ is a wqo w.r.t. single-marked dags. By Higman's Lemma [41], the relation $\trianglerighteq^* \subseteq$ $\mathit{Term}^{\star} \times \mathit{Term}^{\star}$, defined by

$$\langle u^m \rangle \trianglerighteq^* \langle t^n \rangle \Leftrightarrow \exists j^n \left(\begin{array}{c} 1 \leq j_1 < j_2 < \cdots < j_n \leq m \\ \wedge \\ \forall i \in \{ 1 \ldots n \} \, (\, t_i \trianglelefteq_{\mathrm{hom}} u_{j_i} \,) \end{array} \right) \quad , \tag{9.2}$$

is a wqo on tuples of terms. Therefore the infinite sequence $\Delta_0^{A_0}$, $\Delta_1^{A_1}$, $\Delta_1^{A_1}$, ... must contain an infinite subsequence $\Gamma_0^{B_0}$, $\Gamma_1^{B_1}$, $\Gamma_2^{B_2}$, ... such that, for every pair of consecutive tuples $\langle t^n \rangle$ and $\langle u^m \rangle$ in the infinite sequence

$$\langle \langle t^n \rangle \mid i \leftarrow \langle 0 \, , \, 1 \, , \, \ldots \rangle \wedge B_i = \beta^n \wedge (t_{.} = \langle\!| \Gamma_i |\!\rangle_{\beta_{.}})^n \rangle \quad ,$$

it will hold that $\langle u^m \rangle \trianglerighteq^* \langle t^n \rangle$. Hence, it will be the case that

$$\forall \ell < k \in \mathbb{N} \left(\begin{array}{l} B_\ell = \langle \alpha^n \rangle \wedge B_k = \langle \beta^m \rangle \Rightarrow \\ \exists j^n \in \mathbb{N} \left(\begin{array}{c} 1 \leq j_1 < j_2 < \cdots < j_n \leq m \\ \wedge \\ \forall i \in \{ 1 \ldots n \} \exists A \, \left(\Gamma_\ell^{\langle \alpha_i \rangle} \trianglelefteq_{\mathrm{hom}} \Gamma_k^{\langle \beta_{j_i} \rangle} \text{ by } A \right) \end{array} \right) \end{array} \right) \quad .$$

To show that $\langle \mathit{Mdag} \, , \, \preceq \rangle$ is a wqo, it is now sufficient to show that

$$\exists i < j \in \mathbb{N} \, \left(|B_i| = |B_j| \vee \text{varratio } \Gamma_i^{B_i} \leq \text{varratio } \Gamma_j^{B_j} \right) \quad .$$

If there are two dags with the same number of marks, then the above is trivially satisfied. Otherwise, assume that the number of marked nodes is strictly increasing, that is, $|B_i| < |B_j|$ for all $i < j$. It is thus sufficient to show that, for any Γ_i, there cannot be an infinite sequence

$$\Gamma_i^{B_i} \, , \, \Gamma_{i+1}^{B_{i+1}} \, , \, \Gamma_{i+2}^{B_{i+2}} \, , \, \ldots \tag{9.3}$$

such that

$$\text{varratio } \Gamma_i^{B_i} > \text{varratio } \Gamma_{i+1}^{B_{i+1}} > \text{varratio } \Gamma_{i+2}^{B_{i+2}} > \cdots \quad . \tag{9.4}$$

Now, assume that, for all $\Gamma_{i+k}^{B_{i+k}}$, the number of distinct variables is n_k and the number of distinct function symbols is m_k, and thus

$$\text{varratio } \Gamma_{i+k}^{B_{i+k}} = \frac{n_k}{m_k} \quad .$$

With a little algebra it is easy to see that

$$\text{varratio}\ \Gamma_{i+k}^{B_{i+k}} > \text{varratio}\ \Gamma_{i+k+1}^{B_{i+k+1}} \Rightarrow m_k < m_{k+1} \vee n_k > n_{k+1} \ .$$

Since the number of distinct function symbols is bounded, the sequence (9.3) cannot be both infinite and maintain (9.4). Hence, for any $\Gamma_i^{B_i}$, there must be an $\Gamma_j^{B_j}$ such that $i < j$ and $\Gamma_i^{B_i} \preceq \Gamma_j^{B_j}$. □

9.5 Generalisation

As seen in the previous section, $\langle \mathit{Mdag}, \preceq\rangle$ is a well-quasi-order and $\preceq$ can therefore be used to decide when to stop driving. Moreover, $\preceq$ has been designed to allow sufficient unfolding, that is, not to stop prematurely, which I have alluded to in Example 36.

The question is then *what* to do, when I need to stop driving. As described by Turchin [115], I need to *generalise* one of the offending nodes in the process tree, in effect throwing away information that I have acquired during driving. The solution proposed by Sørensen and Glück [104] is to split up such nodes into several parts which can then be explored separately. Generalisations thus give rise to branches in the process tree (as do the speculative unfolding). I will now describe what a generalisation is in the dag world.

The information that $\Gamma^B \preceq \Delta^A$ by C can be used to more than simply to stop driving. The set of cut points C might be used to carve out all trouble spots in the dag Δ^A. In other words, a marked dag Δ^A can be generalised by splitting it *horizontally* in a top and a bottom part. The parts that are carved out of the top part are replaced by fresh variables.

Definition 69 (Horizontal split) For $\Delta^A \in \mathit{Mdag}$ and cut points $B \subseteq \mathit{Addr}$, the *quotient and remainder of* Δ^A *w.r.t.* B, denoted Δ^A/B, is a pair of marked dags defined by

$$\begin{array}{rl} \Delta^{A}/B \overset{\text{def}}{=} \textbf{let} & \bot = \{\, \gamma \mid \beta \in B \wedge \Delta \vdash \beta \overset{*}{\multimap} \gamma \,\} \\ & \top = (\mathscr{D}\Delta) \setminus \bot \\ & \{\, \gamma^{n} \,\} = \{\, \gamma \mid \alpha \in \top \wedge \gamma \in \bot \wedge \Delta \vdash \alpha \multimap \gamma \,\} \\ & \{\, x^{n} \,\} \text{ fresh} \\ & \Gamma_{\top} = \lceil \Delta \rceil_{\top} \uplus \{\, (\gamma. \mapsto x.)^{n} \,\} \\ & \Gamma_{\bot} = \lceil \Delta \rceil_{\bot} \\ \textbf{in} & \left\langle \Gamma_{\top}^{A} \, , \, \Gamma_{\bot}^{\langle \gamma^{n} \rangle} \right\rangle \quad , \end{array}$$

where $\lceil \Delta \rceil_{A}$ means the map Δ restricted to the domain A. □

The quotient and remainder pair can conveniently be expressed as a **let** construct, ensuring that the variables that connect the top and bottom part are kept explicit.

Example 37 Consider again the larger dag Δ shown in the middle of Figure 9.5 and the cut points $\{\, \beta_3 \, , \, \beta_7 \,\}$. The result of $\Delta^{\langle \beta_1 \rangle}/\{\, \beta_3 \, , \, \beta_7 \,\}$ is shown in Figure 9.7, expressed with explicit bindings in a **let** construct. □

Of course, the two dags resulting from a horizontal split together express the same terms as the original dag.

Lemma 37 *If* $\Delta^{A}/B = \left\langle \Gamma_{\top}^{A} \, , \, \Gamma_{\bot}^{\langle \gamma^{n} \rangle} \right\rangle$, *then*

$$\forall \alpha \in A \; (\; \langle\!| \Delta |\!\rangle_{\alpha} = \langle\!| \Gamma_{\top} |\!\rangle_{\alpha} [\, (\langle\!| \Gamma_{\top} |\!\rangle_{\gamma.} := \langle\!| \Gamma_{\bot} |\!\rangle_{\gamma.} \,)^{n} \,] \;) \quad .$$

As mentioned earlier, a dag is generalised so that each part can be driven independently. The point of splitting a dag horizontally is that sharing of variables, and possibly also computations, can be maintained. However, a particular set of cut points might be unfortunate because the horizontal split they induce might *destroy* sharing of variables, and variable sharing is what gives dag-based program transformation the strength that allows me to improve supercompilation, to point where it exhibits some of the results seen in tupling.

Example 38 Let q be the Fibonacci program from Example 31 and ← be the collapsing injection (Figure 8.5). Driving the term '*fib n*' will lead to the

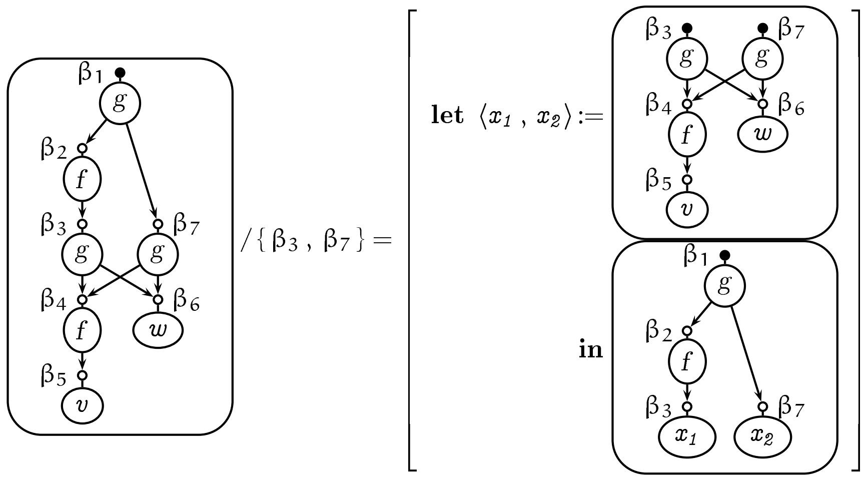
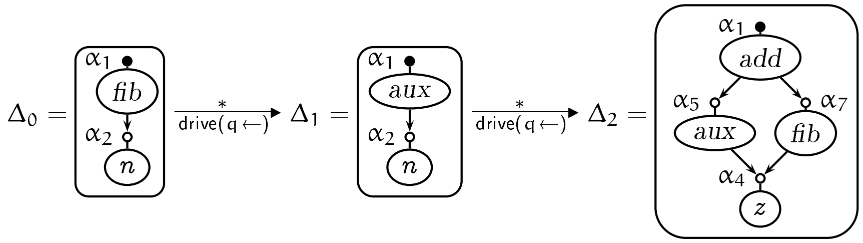
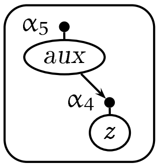


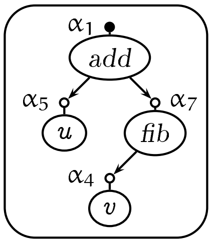
Figure 9.7: Horizontal split

following branch in the process tree:



Since all three dags have only one marked address, it is the case that both $\Delta_0^{\langle\alpha_1\rangle} \preceq \Delta_2^{\langle\alpha_1\rangle}$ by $\{\,\alpha_7\,\}$ and $\Delta_1^{\langle\alpha_1\rangle} \preceq \Delta_2^{\langle\alpha_1\rangle}$ by $\{\,\alpha_5\,\}$. If I choose the latter set of cut points and perform a horizontal split $\Delta_2^{\langle\alpha_1\rangle}/\{\,\alpha_5\,\}$, I will get the unfortunate generalisation

let $\langle u\,,\ v\rangle :=$ 

in 

,

and similarly had I chosen $\Delta_2^{\langle\alpha_1\rangle}/\{\,\alpha_7\,\}$. The problem is that the cut replaces the shared variable z with yet another variable. □

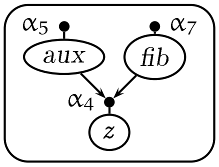
A solution to the loss of variable sharing is to *improve* the set of cut points so that I can avoid splitting shared variables. The function defined below recursively includes more and more parents of variables into the bottom part of the dag, until no more lone variables are to be cut out.

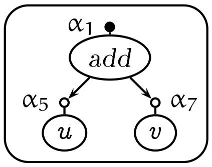
Definition 70 For $\Delta^{A} \in \mathit{Mdag}$ and $B \subseteq \mathit{Addr}$, the *improvement on* B *w.r.t.* Δ^{A} is a set of addresses calculated by improvement B, where

$$\begin{array}{rl}
\text{improvement B} \overset{\text{def}}{=} \textbf{let} & \bot = \{\,\gamma \mid \beta \in B \wedge \Delta \vdash \beta \overset{*}{\multimap} \gamma\,\}\\
& \top = \{\,\gamma \mid \alpha \in A \wedge \Delta \vdash \alpha \overset{*}{\multimap} \gamma\,\} \setminus \bot\\
& I = \{\,\gamma \mid \gamma \in \bot \wedge \alpha \in \top \wedge \Delta \vdash \alpha \multimap \gamma\,\}\\
& V = \{\,\gamma \mid \gamma \in I \wedge \Delta \vdash \gamma \overset{\leadsto}{=} x \in \mathit{Variable}\}\\
& B' = \{\,\alpha \mid \alpha \in \top \wedge \gamma \in V \wedge \Delta \vdash \alpha \multimap \gamma\,\}\\
\multicolumn{2}{r}{\textbf{in if } V = \varnothing \textbf{ then } B \textbf{ else } \text{improvement } (B \cup B')}
\end{array}$$

□

Example 39 Consider again the situation in Example 38, and the unfortunate cut points $\{\,\alpha_5\,\}$. The improvement on $\{\,\alpha_5\,\}$ w.r.t. $\Delta_2^{\langle\alpha_1\rangle}$ is the set $\{\,\alpha_5\,,\ \alpha_7\,\}$, which will lead to the better generalisation

let $\langle u\,,\ v\rangle :=$ 

in 

where the variable z is shared between two computations. □

Lemma 38 *Improvement on cut points terminates.*

PROOF. The set of cut points is non-decreasing in each iteration, so eventually the recursion will stop because the dag is finite. □

There might be situations where the improvement on a set of cut points will result in a *trivial* split, meaning that the top dag consists of variables only, and thus nothing is gained by the operation.

Definition 71 The cut points $C \subseteq$ *Addr* are *trivial* w.r.t. $\Delta^A \in$ *Mdag* iff $\Delta^A/C = \left\langle \Gamma^A_\top , \Gamma^B_\bot \right\rangle$ such that $\forall \alpha \in A$ ($\langle \Gamma_\top \rangle_\alpha \in$ *Variable*). □

If an improved set of cut points turns out to give a trivial split, I can fall back on using the original set of cut points. However, the original cut points might also turn out to be trivial. What to do in such situations depends on whether or not the compared dags have the same number of marks (or roots, as it were).

9.5.1 Same number of marks

When $\Gamma^{\langle \beta^n \rangle} \trianglelefteq_{\text{hom}} \Delta^{\langle \alpha^n \rangle}$ by C, but C is trivial (be it improved or not), it is still possible to make a decent split. The first case is when $\Delta^{\langle \alpha^n \rangle}$ is an *instance* of $\Gamma^{\langle \beta^n \rangle}$.

Definition 72 Let $\Delta^{\langle \alpha^n \rangle}$ and $\Gamma^{\langle \beta^n \rangle} \in$ *Mdag*. I will say that $\Delta^{\langle \alpha^n \rangle}$ *is an instance of* $\Gamma^{\langle \beta^n \rangle}$ *by* θ iff

$$\theta \vdash \Delta^{\langle \alpha^n \rangle} \geq_{\text{inst}} \Gamma^{\langle \beta^n \rangle}$$

can be inferred from the inference system in Figure 9.8. □

That relationship between instances in the dag world and instances in the term world can be formulated precisely.

Lemma 39 *If* $\{ (x_\cdot \mapsto \gamma_\cdot)^m \} \vdash \Delta^{\langle \alpha^n \rangle} \geq_{inst} \Gamma^{\langle \beta^n \rangle}$*, then*

$$\langle \langle \Delta \rangle_{\alpha_1} , \ldots , \langle \Delta \rangle_{\alpha_n} \rangle = \langle \langle \Gamma \rangle_{\beta_1} , \ldots , \langle \Gamma \rangle_{\beta_n} \rangle \, [(x_\cdot {:=} \langle \Delta \rangle_{\gamma_\cdot})^m] \ .$$

$$\Delta \in \mathit{Dag}, \Gamma \in \mathit{Dag}, \alpha \in \mathit{Addr}, \beta \in \mathit{Addr}, \gamma \in \mathit{Addr}, \delta \in \mathit{Addr},$$
$$n \geq 0, \theta \in \mathit{Variable} \Rightarrow \mathit{Addr}$$

$$\frac{}{\varnothing \vdash \Delta^{\langle\rangle} \geq_{\mathrm{inst}} \Gamma^{\langle\rangle}}$$

$$\frac{\Gamma \vdash \beta_0 \stackrel{\rightsquigarrow}{=} x \in \mathit{Variable} \qquad \theta \vdash \Delta^{\langle \alpha^n \rangle} \geq_{\mathrm{inst}} \Gamma^{\langle \beta^n \rangle} \qquad x \notin \mathscr{D}\theta}{\theta \uplus \{ x \mapsto \alpha_0 \} \vdash \Delta^{\langle \alpha_0 \, , \, \alpha^n \rangle} \geq_{\mathrm{inst}} \Gamma^{\langle \beta_0 \, , \, \beta^n \rangle}}$$

$$\frac{\Gamma \vdash \beta_0 \stackrel{\rightsquigarrow}{=} x \in \mathit{Variable} \qquad \theta \vdash \Delta^{\langle \alpha^n \rangle} \geq_{\mathrm{inst}} \Gamma^{\langle \beta^n \rangle} \qquad \langle\!| \Delta |\!\rangle_{\theta x} = \langle\!| \Delta |\!\rangle_{\alpha_0}}{\theta \vdash \Delta^{\langle \alpha_0 \, , \, \alpha^n \rangle} \geq_{\mathrm{inst}} \Gamma^{\langle \beta_0 \, , \, \beta^n \rangle}}$$

$$\frac{\Gamma \vdash \beta_0 \stackrel{\rightsquigarrow}{=} s\ \gamma^m \qquad \Delta \vdash \alpha_0 \stackrel{\rightsquigarrow}{=} s\ \delta^m \qquad s \notin \mathit{Variable} \qquad \theta \vdash \Delta^{\langle \delta^m \alpha^n \rangle} \geq_{\mathrm{inst}} \Gamma^{\langle \gamma^m \beta^n \rangle}}{\theta \vdash \Delta^{\langle \alpha_0 \, , \, \alpha^n \rangle} \geq_{\mathrm{inst}} \Gamma^{\langle \beta_0 \, , \, \beta^n \rangle}}$$

Figure 9.8: Instance-of relation on dags

The instance-of relation is useful because, when $\{ (x_{.} \mapsto \gamma_{.})^m \} \vdash \Delta^{\langle \alpha^n \rangle} \geq_{\mathrm{inst}} \Gamma^{\langle \beta^n \rangle}$, the offending dag $\Delta^{\langle \alpha^n \rangle}$ can be split up into $\Gamma^{\langle \beta^n \rangle}$ and $\Delta^{\langle \gamma^m \rangle}$ using the interface $\langle x^m \rangle$:

$$\textbf{let } \langle x^m \rangle := \Delta^{\langle \gamma^m \rangle} \textbf{ in } \Gamma^{\langle \beta^n \rangle} \ .$$

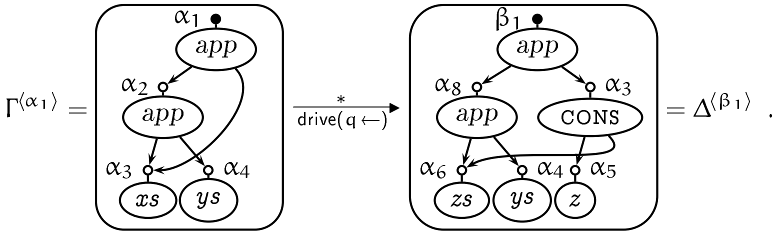
The top dag $\Gamma^{\langle \beta^n \rangle}$ can immediately be folded back to the identical ancestor that caused the generalisation.

The instance-of relation is deterministic, in the sense that if a dag is an instance of another dag, then the map θ is unique.

Lemma 40 $\theta \vdash \Delta^{\langle \alpha^n \rangle} \geq_{\mathit{inst}} \Gamma^{\langle \beta^n \rangle} \wedge \theta' \vdash \Delta^{\langle \alpha^n \rangle} \geq_{\mathit{inst}} \Gamma^{\langle \beta^n \rangle} \Rightarrow \theta = \theta'$.

The second kind of generalisation is the case the offending dag $\Delta^{\langle \alpha^n \rangle}$ has something in common with some ancestor $\Gamma^{\langle \beta^n \rangle}$ (cf. Definition 36), but is not an instance. In that case, the ancestor must be generalised (cf. Figure 5.8).

Example 40 Assume that driving results in the branch



The dag $\Delta^{\langle\beta_1\rangle}$ is not an instance of $\Gamma^{\langle\alpha_1\rangle}$ because the variable *xs* cannot be instantiated to both *zs* and CONS *z zs*. □

What is needed in situations like the one above is a most specific generalisation of the two dag.

Definition 73 Let $\hookleftarrow$ be some realisation of an injection. The *most specific $\hookleftarrow$-generalisation of* $\Gamma^{\langle\beta^n\rangle}$ *w.r.t.* $\Delta^{\langle\alpha^n\rangle}$, denoted msg $\Gamma^{\langle\beta^n\rangle}\ \Delta^{\langle\alpha^n\rangle}$, is a pair of dags $\langle\Gamma_{\top}^{\langle\gamma^n\rangle}, \Gamma_{\perp}^{\langle\delta^m\rangle}\rangle$ derived as follows. Starting with the triple

$$\langle\langle x^n\rangle\ , [\,(x.:=\langle\Delta\rangle_{\alpha.})^n\,]\ , [\,(x.:=\langle\Gamma\rangle_{\beta.})^n\,]\rangle \ ,$$

exhaustively apply the rewrite rule $\xrightarrow[\mathsf{msg}]{}$ (Equation 5.1 on page 93) to derive the most specific term generalisation $\langle\langle t^n\rangle\ , \dots\ , [\,(y.:=u.)^m\,]\rangle$. Assuming without loss of generality that $\mathscr{V}\,\langle t^n\rangle = \langle y^m\rangle$, convert the tuple $\langle t^n\rangle$ into a dag by injecting all the terms into an initial dag $\{\,(\alpha.\mapsto y.)^m\,\}$: Let α^m and γ^n be fresh addresses, and let

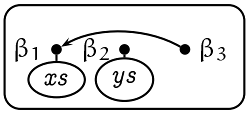
$$\Gamma_{\top}^{\langle\gamma^n\rangle} = \left(\cdots\left(\left(\{\,(\alpha.\mapsto y.)^m\,\}\,[\,\gamma_1\overset{\theta}{\hookleftarrow}t_1\,]\right)[\,\gamma_2\overset{\theta}{\hookleftarrow}t_2\,]\right)\cdots\right)[\,\gamma_n\overset{\theta}{\hookleftarrow}t_n\,] \ ,$$

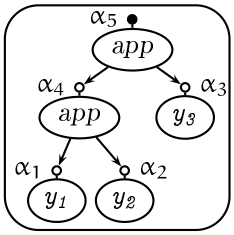
where $\theta = \{\,(y.\mapsto\alpha.)^m\,\}$. Similarly, assuming $\mathscr{V}\,\langle u^m\rangle = \langle z^\ell\rangle$, convert the tuple $\langle u^m\rangle$ into a dag:

$$\Gamma_{\perp}^{\langle\delta^m\rangle} = \left(\cdots\left(\left(\left\{\,(\beta.\mapsto z.)^\ell\,\right\}[\,\delta_1\overset{\theta}{\hookleftarrow}u_1\,]\right)[\,\delta_2\overset{\theta}{\hookleftarrow}u_2\,]\right)\cdots\right)[\,\delta_m\overset{\theta}{\hookleftarrow}u_m\,] \ ,$$

where β^ℓ and δ^m are fresh addresses and $\theta = \left\{\,(z.\mapsto\beta.)^\ell\,\right\}$. □

Example 41 Consider again the dags from Example 40. The most specific generalisation of $\Gamma^{\langle\alpha_1\rangle}$ w.r.t. $\Delta^{\langle\beta_1\rangle}$ is

let $\langle y_1\ ,\ y_2\ ,\ y_3\rangle :=$ 

in 

,

expressed as a **let** construct. □

9.5.2 Different number of marks

When $\Gamma^{\langle\beta^n\rangle} \trianglelefteq_{\text{hom}} \Delta^{\langle\alpha^m\rangle}$ by C, $n < m$ and C is trivial (be it improved or not), it is hard to make a decent split. In lieu of the importance of sharing of variables, an ideal solution would be to separate $\Delta^{\langle\alpha^m\rangle}$ *vertically* into two or more parts such that no sharing of variables is lost. Such a separation could be achieved by placing each mark in a singleton set, and then repeatedly unifying sets if their elements can reach the same variable; should more than one set remain when this procedure reaches a fix-point, a suitable separation has been found.

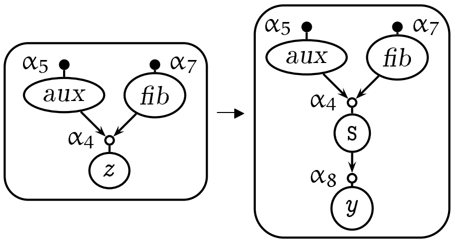
To keep things simple (said he, after having dragged the reader through heaps of idiosyncratic notation), I will simply generalise $\Delta^{\langle\alpha^m\rangle}$ into two dags $\Delta^{\left\langle\alpha_1\ ,\ \dots\ ,\ \alpha_{\lfloor m/2\rfloor}\right\rangle}$ and $\Delta^{\left\langle\alpha_{\lfloor m/2\rfloor+1}\ ,\ \dots\ ,\ \alpha_m\right\rangle}$.

9.6 Fitting it all together

In the previous sections, I have described how to drive dags, how to detect developments that endanger termination by using well-quasi-orders, and how to amend such dangerous situations by generalisations. In this section, I will fit these pieces together to form a dag-based algorithm for supercompilation.

As described in Chapter 5, well-quasi-orders based on homeomorphic embedding are too strict for practical purposes, that is, they tend to force too many generalisations. Especially in the present context, where driving is performed simply by instantiating variables, the well-quasi-order in Definition 68 will immediately blow the whistle after most instantiations.

Example 42 Consider again the driving step

$\Delta^{\langle \alpha_5\ ,\ \alpha_7 \rangle} =$  $= \Gamma^{\langle \alpha_5\ ,\ \alpha_7 \rangle}$.

Clearly $\Delta^{\langle \alpha_5\ ,\ \alpha_7 \rangle} \preceq \Gamma^{\langle \alpha_5\ ,\ \alpha_7 \rangle}$. Generalising $\Gamma^{\langle \alpha_5\ ,\ \alpha_7 \rangle}$ would separate the constructor S from the function symbols *aux* and *fib*.

□

Generalising a dag immediately after a variable has been instantiated would be rather unfortunate, because after an instantiation has taken place, further unfolding can be carried out, effectively performing deforestation on the dag. Moreover, several functions can benefit from the constructors that emerge by the instantiation because of sharing within the dag. It it therefore highly beneficial to be liberal about unfolding as long as there are unstuck redexes in the dag, that is, as long as no instantiations are required. Such liberal unfolding is captured by the deterministic $\xrightarrow[\mathsf{dagx}(\mathsf{q}\,\leftarrow)]{}$ relation (Definition 57).

Another reason to be liberal about deterministic unfolding is that the $\xrightarrow[\mathsf{dagx}(\mathsf{q}\,\leftarrow)]{}$ relation pass over outermost constructors and lone variables, which means that it relieves the dags of uninteresting parts, cleaning them up, as it were.

It therefore seems reasonable to adopt a strategy where each instantiation is followed by exhaustive deterministic unfolding, as long as such deterministic unfolding does not endanger termination — which can be achieved by dividing the nodes in the process tree into classes, according to the way they can be can be unfolded and whether they have already been generalised (cf. Definition 37). The following definition of *relevant ancestors* is prompted by this desire to limit the number of ancestors that should be checked for violations of the well-quasi-order.

Definition 74 Let $\tau \in$ *Tree Mdag* be a transformation trace tree and $\eta \in \mathscr{D}\tau$.

1. η is *generalised* iff it has children and these children are added by means

of a generalisation step. The set of generalised nodes (w.r.t. τ) is denoted *Generalised*.

2. η is *local* iff it is not generalised and $\tau\eta$ can be deterministically unfolded (i.e., $\tau\eta = \Delta^{\langle\beta^n\rangle}$ and $\Delta^{\langle\beta_i\rangle} \xrightarrow[\mathsf{dagx}(q \leftarrow)]{} \Gamma^B$ for some Γ^B). The set of local nodes (w.r.t. τ) is denoted *Local*.

3. η is *global* iff it is neither generalised nor local. The set of global nodes (w.r.t. τ) is denoted *Global*.

4. The set of *relevant ancestors* of a leaf $\eta \in \mathrm{leaves}_\tau$, denoted relanc τ η, is defined as

$$\mathrm{relanc}\ \tau\ \eta \overset{\mathrm{def}}{=} \begin{cases} \{\mu \mid \mu \in \mathit{Global} \wedge \mu <_\tau \eta\}, & \text{if } \eta \in \mathit{Global} \\ \left\{\mu \,\middle|\, \begin{array}{c} \mu \in \mathit{Local} \wedge \\ \forall\nu\ (\mu <_\tau \nu <_\tau \eta \Rightarrow \nu \notin \mathit{Global}) \end{array}\right\}, & \text{if } \eta \in \mathit{Local}. \end{cases}$$

□

The relevant ancestors of a local node is thus limited to other local nodes between itself and the first global node, which ensures that instantiations are not generalised prematurely. Conversely, local nodes are excluded from the relevant ancestors of a global node. Generalised nodes are never considered relevant ancestors, which implies that already generalised nodes never take part generalisation operations, and they are thus generalised only once.

The final ingredient in the supercompilation algorithm is how to administer the process tree, which is made precise in the following definition. The *pending* nodes in the process tree are the nodes that can be driven further and are not renamings of previously seen nodes; they thus need to receive further treatment. Such further treatment is either a generalisation step or a driving step, both of which are performed by updating the process tree by adding a new sequence of children to an existing node, possibly removing existing children in case of a generalisation.

Definition 75 Let $\tau \in$ *Tree Dag* be a process tree.

1. The set of *pending* nodes in τ, denoted $\mathrm{pending}_\tau$, is defined as

$$\mathrm{pending}_\tau \overset{\mathrm{def}}{=} \left\{ \eta \in \mathrm{leaves}_\tau \,\middle|\, \begin{array}{c} \tau\eta \in \mathscr{D}\xrightarrow[\mathsf{drive}(q \leftarrow)]{} \\ \wedge \\ \forall\mu \in (\mathscr{D}\tau \setminus \mathrm{leaves}_\tau)\ (\tau\eta \not\equiv \tau\mu) \end{array} \right\} .$$

2. $\mathrm{addchildren}\,\tau\,\eta\ \langle \Delta_1^{A_1}\,,\ \dots\,,\ \Delta_n^{A_n}\rangle \overset{\mathrm{def}}{=} \tau[\,\eta := (\tau\eta)(\Delta_1^{A_1})\dots(\Delta_n^{A_n})\,]$. □

At last I can present the dag-based transformation algorithm.

Definition 76 The abstract, dag-based supercompilation algorithm $M_{\mathrm{dag}} \in$ *Tree Mdag* ⇨ *Tree Mdag* is shown in Figure 9.9. □

Theorem 4 *M_{dag} terminates.*

PROOF (sketch). To use Lemma 31, I need to show that M_{dag} is Cauchy and that M_{dag} maintains a finitary and continuous predicate $p \in$ *Tree Mdag* ⇨ $\mathbb{B}$.

M_{dag} is Cauchy: Either τ is returned unchanged, or τ is changed by means of addchildren, which by Definition 62 makes M_{dag} Cauchy if I impose the ordering

$$\eta \succ \mu \Leftrightarrow \eta \notin \mathit{Generalised} \wedge \mu \in \mathit{Generalised}\ . \tag{9.5}$$

M_{dag} maintains a finitary and continuous predicate: Let

$$p_{\mathrm{dag}}\ \tau \overset{\mathrm{def}}{=} \begin{cases} \textbf{false}, & \text{if } \exists\{\eta\,,\,\mu\} \subseteq \mathit{Generalised} \\ & \left(\begin{array}{c} \exists i \in \mathbb{N}\ (\ \mu = \eta \mathbin{+\!\!+\!\!-} \langle i\rangle\ )\ \wedge \\ |\langle\!|\tau\eta|\!\rangle| \not\succ |\langle\!|\tau\mu|\!\rangle| \wedge \langle\!|\tau\eta|\!\rangle \not\succ_{\mathrm{inst}} \langle\!|\tau\mu|\!\rangle \end{array}\right) \\ \textbf{false}, & \text{if } \exists\{\eta\,,\,\mu\} \subseteq \mathit{Local} \setminus \mathrm{leaves}_\tau \\ & \left(\begin{array}{c} \eta <_\tau \mu \wedge \tau\eta \preceq \tau\mu \wedge \\ \forall\nu\ (\,\eta <_\tau \nu <_\tau \mu \wedge \nu \notin \mathit{Global}\,) \end{array}\right) \\ \textbf{false}, & \text{if } \exists\{\eta\,,\,\mu\} \subseteq \mathit{Global} \setminus \mathrm{leaves}_\tau \\ & \Big(\eta <_\tau \mu \wedge \tau\eta \preceq \tau\mu\Big) \\ \textbf{true}, & \text{otherwise,} \end{cases}$$

where $\langle\!|\Delta^{\langle\alpha^n\rangle}|\!\rangle \overset{\mathrm{def}}{=} \langle(\langle\!|\Delta|\!\rangle_{\alpha.}\,)^n\rangle$. By Definition 64, p_{dag} is a finitary and continuous predicate. That M_{dag} maintains p_{dag} can be shown by induction on the number of applications of M_{dag} to the initial process tree τ_0. Initially, the process tree consists of a single node, which is neither generalised nor non-leaf, and thus $p_{\mathrm{dag}}\ \tau_0 = \textbf{true}$, as required. For the induction step, I strengthen the induction hypothesis by assuming that non-generalised children μ of a generalised node η

$$\tau \in \mathit{Tree\ Mdag}, \{\alpha, \beta, \gamma\} \subseteq \mathit{Addr}, \{\eta, \mu\} \subseteq \mathbb{N}_1^{\star}$$
$$\{\Delta, \Gamma\} \subseteq \mathit{Mdag}, A \subseteq \mathit{Addr}, B \subseteq \mathit{Addr}, C \subseteq \mathit{Addr}, \{n, m\} \subseteq \mathbb{N}$$

```
$M_{\mathrm{dag}}\ \tau \overset{\mathrm{def}}{=}$
if $\mathrm{pending}_\tau = \varnothing$ then $\tau$
else let $\eta$ suchthat $\eta \in \mathrm{pending}_\tau$
          $\mathit{problems} := \{\mu \in \mathrm{relanc}\ \tau\ \eta \mid \tau\mu \preceq \tau\eta\}$
    in if $\mathit{problems} = \varnothing$
       then if $\eta$ is local
            then let $\Delta^{\langle\alpha^n\rangle} := \tau\eta$
                     $\Gamma^{\langle\beta^m\rangle}$ suchthat $\Delta^{\langle\alpha_i\rangle} \xrightarrow[\mathsf{dagx}(q\leftarrow)]{} \Gamma^{\langle\beta^m\rangle} \wedge i \in \{1 \ldots n\}$
                 in addchildren $\tau\ \eta\ \left\langle \Gamma^{\langle\alpha_1, \ldots, \alpha_{i-1}, \beta^m, \alpha_{i+1}, \ldots, \alpha_n\rangle} \right\rangle$
            else let $\{\Delta_1^{A_1}, \ldots, \Delta_n^{A_n}\} := \left\{ \Delta^A \,\middle|\, \tau\eta \xrightarrow[\mathsf{drive}(q\leftarrow)]{} \Delta^A \right\}$
                in addchildren $\tau\ \eta\ \langle \Delta_1^{A_1}, \ldots, \Delta_n^{A_n} \rangle$
       else let $\mu$ suchthat $\mu \in \mathit{problems}$
                $\Gamma^B := \tau\mu$
                $\Delta^A := \tau\eta$
                $C$ suchthat $\Gamma^B \trianglelefteq_{\mathrm{hom}} \Delta^A$ by $C$
                $C' :=$ improvement $C$
          in if $C'$ is nontrivial
             then addchildren $\tau\ \eta\ (\Delta^A/C')$
             else if $C$ is nontrivial
                  then addchildren $\tau\ \eta\ (\Delta^A/C)$
                  else if $|A| = |B|$
                       then if $\{(x. \mapsto \gamma.)^m\} \vdash \Delta^A \geq_{\mathrm{inst}} \Gamma^B$
                            then addchildren $\tau\ \eta\ \left\langle \Gamma^B, \Delta^{\langle\gamma^m\rangle} \right\rangle$
                            else addchildren $\tau\ \mu\ (\mathrm{msg}\ \Gamma^B\ \Delta^A)$
                       else let $\langle\alpha^n, \beta^m\rangle := A$ suchthat $|n - m| \leq 1$
                           in addchildren $\tau\ \eta\ \left\langle \Delta^{\langle\alpha^n\rangle}, \Delta^{\langle\beta^m\rangle} \right\rangle$
```

Figure 9.9: Supercompilation using dags. The algorithm is presented as an abstract program transformer that does a single step of transformation with each invocation.

do not match the first case of p_{dag}, that is,

$$\forall\eta \in \mathit{Generalised}, \mu \in \mathrm{leaves}_\tau \left(\begin{array}{c} \exists i \; (\; \mu = \eta +\!\!\!+ \langle i \rangle \;) \\ \Rightarrow \\ |\langle\tau\eta\rangle| > |\langle\tau\mu\rangle| \vee \langle\tau\eta\rangle >_{\mathrm{inst}} \langle\tau\mu\rangle \end{array} \right) \quad , \qquad (9.6)$$

as well as $p_{\mathrm{dag}}\ \tau = \mathbf{true}$. I proceed by case analysis (cf. Figure 9.9).

If *problems* $= \varnothing$, then a new process tree τ' is produced by adding a sequence of new children to η, and thus $\eta \notin$ *Generalised* w.r.t. τ, which ensures that the first case of p_{dag} does not match. Since *problems* $= \varnothing$ implies that no relevant ancestors of η are $\preceq$-related to η, neither the second nor third case of p_{dag} matches. Hence $p_{\mathrm{dag}}\ \tau' = \mathbf{true}$. Since moreover $\eta \notin$ *Generalised*, τ' satisfies Equation 9.6.

Otherwise, assume $\mu \in$ *problems*, node η is labelled Δ^A, node μ is labelled Γ^B, and $\Gamma^B \trianglelefteq_{\mathrm{hom}} \Delta^A$ by C. If C' (or C) is nontrivial, then a new tree τ' is produced by generalising node η by adding new children. Since C' (respectively C) is nontrivial, both the new children resulting from the horizontal split Δ^A/C' (respectively Δ^A/C) are smaller than Δ^A. Hence τ' satisfies the induction hypothesis.

Otherwise, assume $|A| = |B|$. If Δ^A is an instance of Γ^B, then η is generalised in the new tree τ' by adding Γ^B and some part of Δ^A as children. Since $\langle\Delta^A\rangle >_{\mathrm{inst}} \langle\Gamma^B\rangle$ and the part of Δ^A is strictly smaller than Δ^A (unless $\langle\Gamma^B\rangle = \langle x \rangle$, which is not hard to show is impossible), τ' satisfies the induction hypothesis. Otherwise, τ' is obtained by replacing the subtrees of μ with two dags that constitute the most specific generalisation of Δ^A andΓ^B. Since C is trivial and Δ^A is not an instance of Γ^B, it is not hard to show that the two new added to τ' does not violate Equation 9.6.

Otherwise, assume $|A| \neq |B|$. Since $\Gamma^B \trianglelefteq_{\mathrm{hom}} \Delta^A$ by C, it must be the case that $|A| > |B| \geq 1$. Thus, producing a new tree τ' by splitting up Δ^A into two dags of comparable number of marks and adding these dags as children to η does not violate Equation 9.6. □

9.7 Example of dag-based supercompilation

You can observe some of the interesting effects that dag-based supercompilation causes by inspecting what happens when I apply M_{dag} to the well-known

Fibonacci program (see Example 31). If I start out with the term *fib* n encoded as a dags, and repeatedly apply $\mathsf{M}_{\mathsf{dag}}$ using the collapsing injection, I will get a process tree very similar to the one depicted in Figures 9.10, 9.11 and 9.12.

The program that can be extracted from the process tree is

$$\begin{aligned}
g_{\langle\rangle}\ \mathsf{0} &= \mathbf{let}\ \langle u\rangle = \mathsf{0}\ \mathbf{in}\ \mathsf{s}\ u\\
g_{\langle\rangle}\ (\mathsf{s}\ z) &= g_{\langle 21\rangle}\ z\\
g_{\langle 21\rangle}\ \mathsf{0} &= \mathbf{let}\ \langle u\rangle = \mathsf{0}\ \mathbf{in}\ \mathsf{s}\ u\\
g_{\langle 21\rangle}\ (\mathsf{s}\ z) &= \mathbf{let}\ \langle u\ ,\ v\rangle = g_{\langle 212111\rangle}\ z\ \ \mathbf{in}\ g_{\langle 212112\rangle}\ u\ v\\
g_{\langle 212111\rangle}\ \mathsf{0} &= \mathbf{let}\ \langle s\ ,\ t\rangle = \langle \mathsf{0}\ ,\ (\mathbf{let}\ \langle u\rangle = \mathsf{0}\ \mathbf{in}\ \mathsf{s}\ u)\rangle\ \ \mathbf{in}\ \langle(\mathsf{s}\ s)\ ,\ t\rangle\\
g_{\langle 212111\rangle}\ (\mathsf{s}\ y) &= \mathbf{let}\ \langle u\ ,\ v\rangle = g_{\langle 212111\rangle}\ y\\
&\quad\ \mathbf{in}\ \mathbf{let}\ z = g_{\langle 212112\rangle}\ u\ v\ \mathbf{in}\ \langle z\ ,\ u\rangle\\
g_{\langle 212112\rangle}\ \mathsf{0}\ \ v &= v\\
g_{\langle 212112\rangle}\ (\mathsf{s}\ w)\ v &= \mathbf{let}\ z = g_{\langle 212112\rangle}\ w\ v\ \mathbf{in}\ \mathsf{s}\ z
\end{aligned}$$

The above program contains a lot of superfluous **let** constructs resulting from generalisations or the CONS rule. These **let** constructs can be removed by simple inlining, resulting in the following program.

$$\begin{aligned}
g_{\langle\rangle}\ \mathsf{0} &= \mathsf{s}\ \mathsf{0}\\
g_{\langle\rangle}\ (\mathsf{s}\ z) &= g_{\langle 21\rangle}\ z\\
g_{\langle 21\rangle}\ \mathsf{0} &= \mathsf{s}\ \mathsf{0}\\
g_{\langle 21\rangle}\ (\mathsf{s}\ z) &= \mathbf{let}\ \langle u\ ,\ v\rangle = g_{\langle 212111\rangle}\ z\ \ \mathbf{in}\ g_{\langle 212112\rangle}\ u\ v\\
g_{\langle 212111\rangle}\ \mathsf{0} &= \langle(\mathsf{s}\ \mathsf{0})\ ,\ (\mathsf{s}\ \mathsf{0})\rangle\\
g_{\langle 212111\rangle}\ (\mathsf{s}\ y) &= \mathbf{let}\ \langle u\ ,\ v\rangle = g_{\langle 212111\rangle}\ y\ \ \mathbf{in}\ \big\langle(g_{\langle 212112\rangle}\ u\ v)\ ,\ u\big\rangle\\
g_{\langle 212112\rangle}\ \mathsf{0}\ \ v &= v\\
g_{\langle 212112\rangle}\ (\mathsf{s}\ w)\ v &= \mathsf{s}\ (g_{\langle 212112\rangle}\ w\ v)
\end{aligned}$$

The remaining **let** constructs cannot be eliminated by simple inlining because they serve a specific purpose: They cache the intermediate results, as described in Section 2.4.2, avoiding unnecessary re-computation. But since my little language is not equipped with **let** constructs, I will have to encode these constructs (cf. page 165). In the case of the above program, I can simply intro-

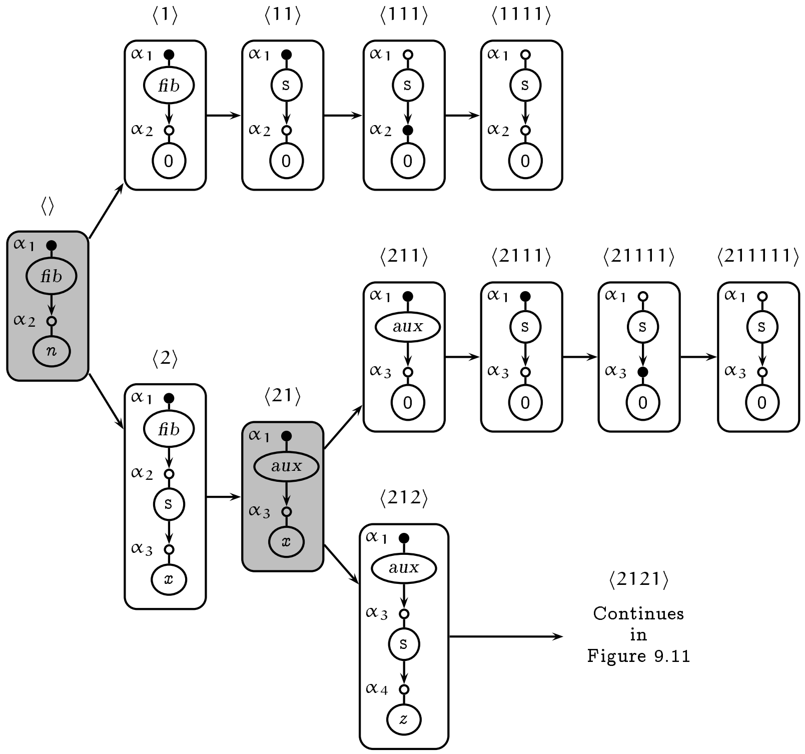


Figure 9.10: Transformation of Fibonacci using dags. Global nodes are shaded.

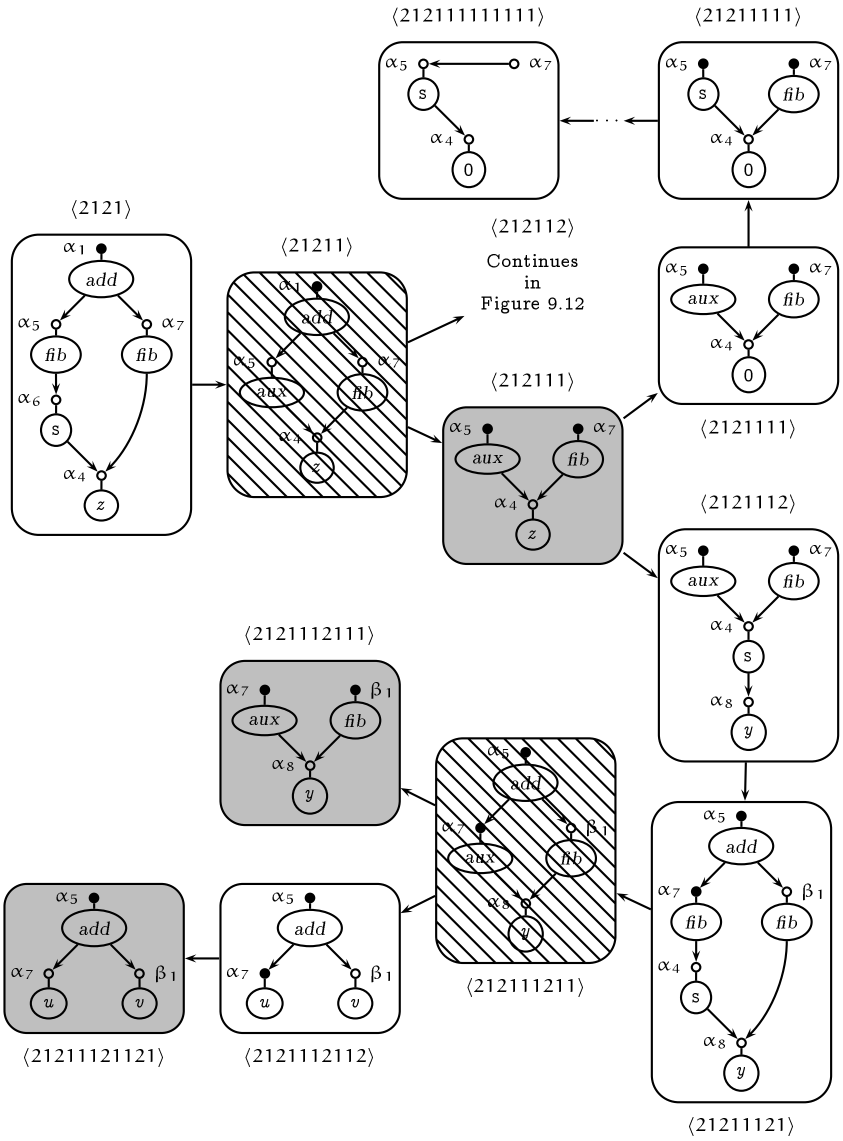


Figure 9.11: Transformation of Fibonacci using dags (cont.). Global nodes are shaded and generalised nodes are dashed out.

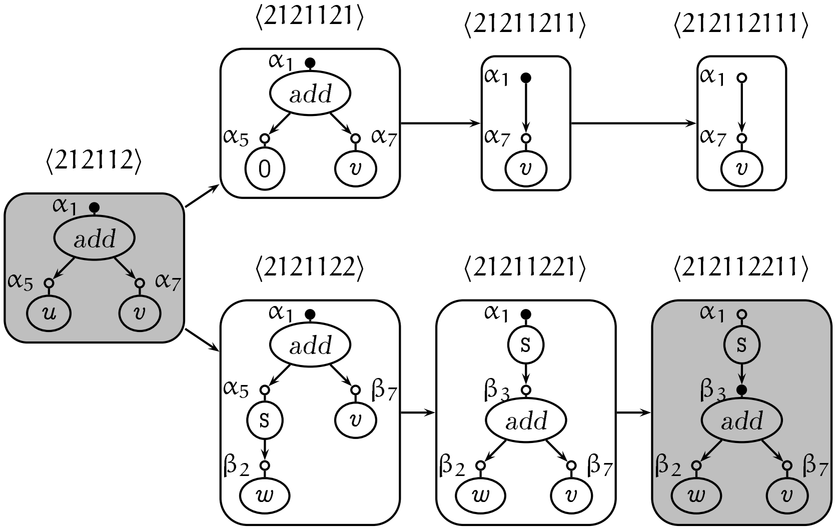


Figure 9.12: Transformation of Fibonacci using dags (cont.). Global nodes are shaded.

duce a two-tuple type and use pattern matching to deconstruct the tuples.

$$\begin{array}{ll}
\mathbf{data}\ \mathrm{Tuple2}\ a\ y & = \texttt{T2}\ a\ b \\
g_{\langle\rangle}\ \texttt{0} & = \texttt{S}\ \texttt{0} \\
g_{\langle\rangle}\ (\texttt{S}\ z) & = g_{\langle 21\rangle}\ z \\
g_{\langle 21\rangle}\ \texttt{0} & = \texttt{S}\ \texttt{0} \\
g_{\langle 21\rangle}\ (\texttt{S}\ z) & = \mathit{split}_{\langle 21211\rangle}\ (g_{\langle 212111\rangle}\ z) \\
\mathit{split}_{\langle 21211\rangle}\ (\texttt{T2}\ u\ v) & = g_{\langle 212112\rangle}\ u\ v \\
g_{\langle 212111\rangle}\ \texttt{0} & = \langle(\texttt{S}\ \texttt{0})\ ,\ (\texttt{S}\ \texttt{0})\rangle \\
g_{\langle 212111\rangle}\ (\texttt{S}\ y) & = \mathit{split}_{\langle 212111211\rangle}\ (g_{\langle 212111\rangle}\ z) \\
\mathit{split}_{\langle 212111211\rangle}\ (\texttt{T2}\ v\ v) & = \texttt{T2}\ (g_{\langle 212112\rangle}\ u\ v)\ u \\
g_{\langle 212112\rangle}\ \texttt{0}\ \ v & = v \\
g_{\langle 212112\rangle}\ (\texttt{S}\ w)\ v & = \texttt{S}\ (g_{\langle 212112\rangle}\ w\ v)
\end{array}$$

In general, however, I must be careful not to assume that a function which is supposed to construct a tuple indeed do so, because, if the function is non-terminating, it may not produce the outermost tuple constructor needed for the pattern match.

Remark 13 In the context of supercompilation (and program transformation in general), non-termination is not merely an academic problem, since the supercompiler might *introduce* non-termination, albeit in a harmless way. The problem is that some branches of the process tree might be unreachable, in the sense that the program cannot reach that particular state, no matter what arguments you give to it — and this state may very well turn out to be non-terminating. This imprecision has two sources. Firstly, most generalisation steps will effectively lose information separating a dag into two disjoint parts. Secondly, the driving mechanism presented in this chapter only propagates positive information (cf. Chapters 5 and 6).

The general solution for eliminating tuples is to introduce tuple constructors together with projection functions, and pass the tuples through functions that extract the tuple elements by means of the projection functions, which ensures that the tuple (as well as other variables) can be shared and that the extraction of the elements will be carried out when, and only when, the elements are needed. For the preceding program, general tuple elimination would produce the following program.

$$
\begin{array}{ll}
\textbf{data}\ \mathrm{Tuple2}\ a\ y & = \mathrm{T2}\ a\ b \\
\mathit{prj1of2}\ (\mathrm{T2}\ x\ y) & = x \\
\mathit{prj2of2}\ (\mathrm{T2}\ x\ y) & = y \\
g_{\langle\rangle}\ 0 & = \mathrm{S}\ 0 \\
g_{\langle\rangle}\ (\mathrm{S}\ z) & = g_{\langle 21\rangle}\ z \\
g_{\langle 21\rangle}\ 0 & = \mathrm{S}\ 0 \\
g_{\langle 21\rangle}\ (\mathrm{S}\ z) & = \mathit{split}_{\langle 21211\rangle}\ (g_{\langle 212111\rangle}\ z) \\
\mathit{split}_{\langle 21211\rangle}\ \mathit{tup} & = g_{\langle 212112\rangle}\ (\mathit{prj1of2}\ \mathit{tup})\ (\mathit{prj2of2}\ \mathit{tup}) \\
g_{\langle 212111\rangle}\ 0 & = \langle(\mathrm{S}\ 0)\ ,\ (\mathrm{S}\ 0)\rangle \\
g_{\langle 212111\rangle}\ (\mathrm{S}\ y) & = \mathit{split}_{\langle 212111211\rangle}\ (g_{\langle 212111\rangle}\ z) \\
\mathit{split}_{\langle 212111211\rangle}\ \mathit{tup} & = \mathrm{T2}\ (g_{\langle 212112\rangle}\ (\mathit{prj1of2}\ \mathit{tup})\ (\mathit{prj2of2}\ \mathit{tup}))\ (\mathit{prj1of2}\ \mathit{tup}) \\
g_{\langle 212112\rangle}\ 0\ \ v & = v \\
g_{\langle 212112\rangle}\ (\mathrm{S}\ w)\ v & = \mathrm{S}\ (g_{\langle 212112\rangle}\ w\ v)
\end{array}
$$

9.8 Summary and Related work

The benefits of *deforestation* and *supercompilation* are well illustrated in the literature, and advances in ensuring termination of these (and similar) transformations have greatly improved their potential as *automatic*, off-the-shelf optimisation techniques. One problem, however, remains in making these driving-based techniques suitable for inclusion in the standard tool-box employed by compiler writers: It is in general not possible to ensure that a transformed program is at least as efficient as the original program, without imposing severe (usually syntactic) restrictions on the original programs.

In this chapter, I have formulates a version of *positive supercompilation* that tries to address the concern of ensuring efficiency of the transformed program. The key ingredient in this formulation is that it is based on a graph representation of terms where both driving and generalisations are performed directly on dags. I have presented a version of supercompilation that — when using collapsed jungles as the underlying representation — can give some of the speedups that collapsed-jungle evaluation can give, but without any run-time overhead. In this respect, I have effectively achieved to perform *tupling* (Pettorossi [81] and Chin [16], cf. Chapter 7. The key ingredient in tupling is to discover a set of *progressive cuts* [81] in the call graph for a program, and automatic search procedures for such cuts have been investigated intensively. In particular, Pettorossi, Pietropoli and Proietti [82] manipulate dags in a fashion that is similar to my notion of an improved split. However, their paper is concerned with parallelisation of extremely restricted programs (more precisely, direct-recursion, single-exit functions). In contrast to call-graph-based approaches, I am able to *synchronise* common calls, because I use a local/global unfolding strategy similar to what is used in *partial deduction* (see e.g., Gallagher [34] or De Schreye et al [26]). A crucial ingredient in the dag-based transformation is thus that all local unfoldings are carried out before any instantiations are performed (as long as termination is not endangered).

The separation between global and local unfolding is based on ideas from Partial Deduction, but was already present in the work Turchin [115] (he calls local unfolding for *transient reductions*). Formalisation of local unfoldings in positive supercompilation was introduced by Sørensen [106]. In the present formulation, global unfoldings are not really unfoldings, but rather speculative execution by means of instantiations: The instantiations make local unfoldings possible.

The use of wqo's in a program-transformation settings seems to have started around 1993. In Partial Deduction (partial evaluation of logic programs), wqo's were apparently first used by Bol [9] and Sahlin [92], whereas Sørensen [103] seems to have been the first to use wqo's for supercompilation. The strength of wqo's have further been investigated by Leuschel [64].

Further work needs to be done in three directions. Firstly, I have not provided a proof correctness of the transformation, in the sense that the transformed program is semantically equivalent with the original, but I have provided a notion of observability (see Chapter 3) that is needed to carry out such a proof. The major obstacle in such a proof is how treat fold notes in the process tree (i.e., recursion in the transformed program).

Secondly, it is hard to characterise which programs do not lead to a loss in efficiency. The main obstacle is that generalisations are performed on-line, and therefore the operations which separate a dag into disjoint parts are highly dynamic in nature. It would be interesting to devise an off-line analysis that would only accept programs for which non-degrading efficiency could be guaranteed.

Thirdly, It would be interesting to shift the focus from the rather theoretical question about optimisation of whole programs to the more pragmatic question of optimising *interfaces between modules*. Modules should be transformed w.r.t. an interface, which in my case would be a set of terms that describe the intended or actual usage of the module. Such a shift of focus would have a flavour of real software-engineering, in which programs are constructed from *generic components*. When such components are fitted together, the overhead induced by interfaces and unnecessary generality can be removed.

Chapter 10

Implementation

In this chapter I will give an overview of an implementation of the dag-based system that I described in the previous chapter, and I will run some of the preceding examples through the system and compare the transformed programs to their original counterparts.

The system consists of three parts.

- An **unparser** that transforms the internal object-program representation into human-readable text format.

- An **interpreter** for the object language. The interpreter uses standard dags for term representation (cf. Section 8.2.1), that is, it emulates call-by-need, and it can output number of rewrite steps as well as number of memory cells used.

- A **supercompiler** that uses fully-collapsed dags as the underlying term representation (cf. Section 8.2.3). It can produce a graphical representation of the process tree.

All parts of the system has been implemented using the excellent Moscow ML system[1] created by Sestoft, Russo and Romanenko.

[1] `http://www.dina.kvl.dk/~sestoft/mosml.html`

10.1 Object language

To make programming in the object language somewhat more practical, the language treated by the implementation is a little different from the one described in previous chapters. First of all, I have added primitive types for booleans and natural numbers, and with these types a number of standard primitive operators like $+$ and $\leq$.

Secondly, all pattern matching definitions are required to have a *catch-all* pattern (e.g., $g\ x\ y^n \overset{q}{=} t$), which is used when non of the other patterns match, and the special term `error` can be used to abort evaluation of the program. There is therefore no distinction between f-functions and g-functions in the object language.

Thirdly, a primitive construct for conditional statements together with two non-standard boolean operators have been added to the language. The boolean connectives are binary-and (`&`) and binary-or (`|`), and they evaluate their arguments in a *fair* manner. In other words, if we let $\perp$ symbolise an non-terminating computation and let t be any term of boolean type, the following rules hold for `&` and `|`:

$$\begin{array}{rcccl@{\qquad\qquad}rcccl}
\mathbf{true} & \texttt{\&} & t & = & t & \mathbf{true} & | & t & = & \mathbf{true} \\
\mathbf{false} & \texttt{\&} & t & = & \mathbf{false} & \mathbf{false} & | & t & = & t \\
\perp & \texttt{\&} & t & = & t & \perp & | & t & = & t \\
t' & \texttt{\&} & t & = & t \texttt{ \& } t' & t' & | & t & = & t \mid t'
\end{array} \tag{10.1}$$

Fairness is implemented by swapping the arguments after each step of evaluation. The fairness of boolean connectives makes program transformation insensitive to the order in which conditions are written, a property that, in my experience, has proven to be valuable in program specialisation in general, and in deriving *checkers* from inference algorithms in particular (cf. Chapters 11 and 12).

10.2 Dags and homeomorphic embedding

The supercompiler uses fully-collapsed dags as the underlying term representation. As discussed in Chapter 9, the homeomorphic embedding relation and variations thereof are rather expensive to compute, and therefore I use one large dag to represent all terms that surface during transformation, and I cache homeomorphic relations between sub-dags. More precisely, the process tree contains references into one large fully-collapsed dag, and during transformation I maintain a dynamic table (using Sleator and Tarjan's *splay trees* [101]) holding all previously calculated relations between sub-dags, like suggested in the proof of Lemma 35 on page 177. The process tree itself represented upside-down, in the sense that the tree is a list of leaves, each pointing to its immediate predecessor, and arrangement that makes it easy to add new leaves.

Since every sub-dag can potentially be shared among several nodes in the process tree, I cannot simply rewrite redexes *in place*. Instead, after having created a contractum, I rebuild all parts of the current sub-dag which reach the redex (the *spine*), such that they afterwards reach the contractum instead of the redex.

With respect to maintaining a fully-collapsed dag, the search for an existing node, which in Figure 8.5 on page 143 is formulated as

$$\exists\beta \in \mathscr{D}\Delta \ \textbf{suchthat} \ \Delta\beta = s\ \alpha^n \ ,$$

is implemented by collecting a set P of all parents of α_1 that are labelled s, then removing all elements which have not α_2 as second child from P, and so forth, until either P is empty or a node $s\ \alpha^n$ is found. If the sought node has no children (i.e., $n = 0$), I lookup the node in a cache containing all 0-ary symbols present in the dag.

10.3 Examples

10.3.1 Double append

Supercompiling the list-append program

```
data List x = N | C x (List x)
app N x = x
app (C x y) z = C x (app y z)
```

together with the term `app (app x y) z` produces the program

```
data List x = N | C x (List x)
match0 N x y = match4 x y
match0 (C x y) z v = C x (match0 y z v)
```

```
match4 N x = x
match4 (C x y) z = C x (match4 y z)
main x y z = match0 x y z
```

in which the intermediate list construction has been eliminated.

10.3.2 String match

Supercompiling the list-matcher program

```
data List x = N | C x (List x)
match x y = chkpat x y x y
chkpat N x y z = true
chkpat (C x y) z v w = chkstr z x y v w
chkstr N x y z v = false
chkstr (C x y) z v w x1 = if x == z then chkpat v y w x1 else next x1 w
next (C x y) z = match z y
```

together with the term `match (C 1 (C 1 (C 2 N))) x` produces the program

```
data List x = N | C x (List x)
match2 N = false
match2 (C x y) = cond7 x y
cond7 x y = if x == 1 then match11 y else match2 y
match11 N = false
match11 (C x y) = cond16 x y
cond16 x y = if x == 1 then match20 y else cond7 x y
match20 N = false
match20 (C x y) = if x == 2 then true else cond16 x y
main x = match2 x
```

in which the list $[1, 1, 2]$ has been hardwired into the control structure. As you can see by the call to `cond7` in the `else`-branch in `cond6`, there is no propagation of negative information.

10.3.3 Fibonacci

Supercompiling the Fibonacci program

```
data Nat = Z | S Nat
add Z x = x
add (S x) y = S (add x y)
fib Z = S Z
fib (S x) = aux x
```

```
aux Z = S Z
aux (S x) = add (aux x) (fib x)
main x = fib x
```

produces a new program in which repeated computations have been tupled together:

```
data Tuple2 x y = T2 x y
data Nat = Z | S Nat
match0 Z = S Z
match0 (S x) = match5 x
match5 Z = S Z
match5 (S x) = decon10 (match12 x)
decon10 (T2 x y) = match11 x y
match12 Z = T2 (S Z) (S Z)
match12 (S x) = decon30 (match12 x)
decon30 (T2 x y) = match31 x y
match31 Z x = T2 x Z
match31 (S x) y = T2 (S (match11 x y)) (S x)
match11 Z x = x
match11 (S x) y = S (match11 x y)
main x = match0 x
```

10.3.4 Specialisation of an interpreter

Consider a small functional language with the abstract syntax

$$
\begin{array}{llcclr}
\text{Function} & \texttt{Fun} & \ni & f & ::= & \lambda x^n.e & \text{(abstraction)} \\
\text{Expression} & \texttt{Exp} & \ni & e & ::= & x & \text{(variable)} \\
 & & & & \mid & c & \text{(integer)} \\
 & & & & \mid & e + e \mid e - e \mid e * e & \text{(arithmetic)} \\
 & & & & \mid & \textbf{let } x := e \textbf{ in } e & \text{(binding)} \\
 & & & & \mid & \textbf{if } e \neq 0 \textbf{ then } e \textbf{ else } e & \text{(conditional)} \\
 & & & & \mid & f\ e^n & \text{(function call)}
\end{array}
$$

where a program is a list of functions. An interpreter for the above language is shown in Figure 10.1. The interpreter encodes a function definition '$f = \lambda x^n.e$' as '`Fn` f $[x_1...x_n]$ e', where the names are represented as integers, that is, f is a unique integer and $[x_1...x_n]$ is a list of integers, numbering the formal parameters. Accordingly, variables and function names in expressions are also represented as integers. As an example, consider the factorial function

$$fac = \lambda x\,.\,\textbf{if } x \neq 0 \textbf{ then } x * (fac\ (x - 1)) \textbf{ else } 1\ ,$$

```
data List x = N | C x (List x)
data Exp = Var Int
         | Cst Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Let Int Exp Exp
         | Con Exp Exp Exp
         | App Int (List Exp)
data Fun = Fn Int (List Int) Exp
data Env = Emp | Bnd Int Int Env
lkbody (C x y) z = matchbody x y z
matchbody (Fn x y z) v w = if x == w then z else lkbody v w
lkpars (C x y) z = matchpars x y z
matchpars (Fn x y z) v w = if x == w then y else lkpars v w
lkvar (Bnd x y z) v = if x == v then y else lkvar z v
bind (C x y) z v w x1 = bnd z x y v w x1
bind x y z v w = z
bnd (C x y) z v w x1 x2 = Bnd z (eval x x1 x2) (bind v y w x1 x2)
eval (Var x) y z = lkvar y x
eval (Cst x) y z = x
eval (Add x y) z v = (eval x z v) + (eval y z v)
eval (Sub x y) z v = (eval x z v) - (eval y z v)
eval (Mul x y) z v = (eval x z v) * (eval y z v)
eval (Let x y z) v w = eval z (bind (C x N) (C y N) v v w) w
eval (Con x y z) v w = if (eval x v w) == 0 then eval z v w else eval y v w
eval (App x y) z v = eval (lkbody v x) (bind (lkpars v x) y Emp z v) v
run x y = eval (lkbody x 0) (bind (lkpars x 0) (C (Cst y) N) Emp Emp x) x
```

Figure 10.1: Interpreter for a small functional language.

```
facpgm = C (Fn 0 (C 1 N)
                 (Con (Var 1)
                      (Mul (Var 1) (App 0 (C (Sub (Var 1) (Cst 1)) N)))
                      (Cst 1)))
           N
```

Figure 10.2: Factorial program.

which can be encoded as the program seen in Figure 10.2. The recursive function has number 0 and the formal parameter has number 1.

The interpreter in Figure 10.1 is invoked through the run function, which, when provided a program and a single integer, evaluates the body of the 0th function in the program with 0th argument bound to the integer. The eval function evaluates expressions in a call-by-value fashion, using the bind function to bind variables to values in the environment.

Transforming the interpreter together with the facpgm gives the following program.

```
data List x = N | C x (List x)
data Exp = Var Int | Cst Int | Add Exp Exp | Sub Exp Exp | Mul Exp Exp
         | Let Int Exp Exp | Con Exp Exp Exp | App Int (List Exp)
data Fun = Fn Int (List Int) Exp
data Env = Emp | Bnd Int Int Env
cond101 x y =
  if (x - 1) == 0 then 1 else (x - 1) * (cond101 (x - 1) (Bnd 1 x Emp))
main x = if 0 == x then 1 else x * (cond101 x Emp)
```

As you can see, the interpretive overhead has been removed. The superfluous second parameter of the cond101 function is the remains of the environment used to bind variables in the interpreter; it could be removed by a post-processing phase based on abstract interpretation, see Chin [14, Chapter 2] or Leuschel and Sørensen [67]. Also, the superfluous data-type definitions could easily be removed by simple post-processing phase.

10.3.5 Conclusion

All of the previous examples have been run through the system on a 1.5GHz Intel Pentium machine with 256MB memory. The results are shown in Table 10.1.

Table 10.1: Running times in μs, averaged over a 100 runs.

Transformation			Interpretation			
Program	Term	Time	Input	Orig	Trans	Speedup
Append	app (app x y) z	11.8	[3,4,5] [1,1,2] [1,1,1,2]	9.4	9.1	3.2%
Listmatch	match [1,1,2] x	73.6	[1,1,1,2]	9.2	6.5	29.3%
Fibonacci	fib x	19.8	8	6.3	8.0	-27.0%
			10	329.2	54.0	83.6%
			12	4234.2	286.6	93.2%
Interpreter	run facpgm x	7433.6	10	62.6	14.9	76.2%

From the table it can be seen that the transformed programs are faster than their original counterparts, except for the Fibonacci program when run on small numbers. The slowdown stems from the construction and destruction of tuples used in the transformed program.

Chapter 11

Program inversion

Abramov and Glück have recently introduced a technique called URA for inverting first order functional programs. Given some desired output value, URA computes a potentially infinite sequence of substitutions/restrictions corresponding to the relevant input values. In some cases this process does not terminate.

In this chapter, I propose a new program analysis for inverting programs. The technique works by computing a finite grammar describing the set of all input that relate to a given output. During the production of the grammar, the original program is implicitly transformed using driving steps. Whereas URA is sound and complete, but often fails to terminate, my technique always terminates and is complete, but not sound. As an example, I demonstrate how to derive type inference from type checking.

The idea of approximating functional programs by grammars is not new. For instance, Sørensen [105] has developed a technique using tree grammars to approximate termination behaviour of deforestation. However, for the present purposes it has been necessary to invent a new type of grammar that extends tree grammars by permitting a notion of sharing in the productions. These *dag grammars* seem to be of independent interest.

11.1 Introduction

The program-transformation techniques collectively called *supercompilation* have been shown to effectively handle problems that partial evaluation and

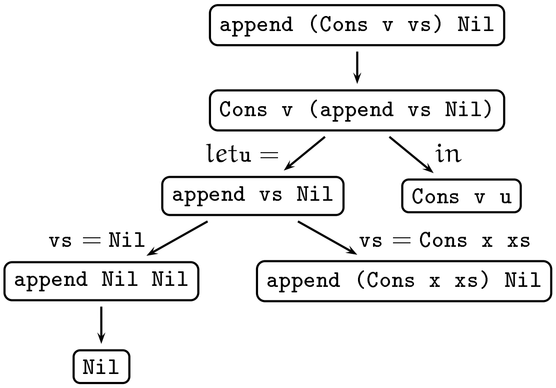


Figure 11.1: Process tree

deforestation cannot handle. The apparent strength of supercompilation stems from *driving* the object programs, that is, speculatively unfolding expressions containing variables, based on the possible executions described by the program. As an example of driving, consider the Haskell-like program

```
data List x = Nil | Cons x (List x)
main v vs = append (Cons v vs) Nil
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

where, by convention, the `main` definition serves as the interface to the program. It is not possible to execute this program because of the variables `v` and `vs` in `main v vs`, but I can *drive* the program, resulting in a *process tree* describing all possible computations of the program. For the above program, one possible process tree is shown in Figure 11.1.

In general, the process tree is constructed as follows. The root of the process tree is labelled with right-hand side of the `main` definition. New leaves are added to the tree repeatedly by inspecting the label of some leaf, and either

1. *Unfolding* an outermost call (①→②, ⑥→⑦).

2. *Instantiating* a variable that hinders unfolding (④→⑤⑥).

3. *Generalising* by creating two disjoint labels (②→③④).

Whenever the label of a leaf is identical to the label of an ancestor (up to renaming of variables), that leaf is not unfolded further (⑤--→①).

A new (slightly more efficient) program can easily be extracted from a process tree, namely

```
data List x = Nil | Cons x (List x)
main v vs = Cons v (appendnil vs)
appendnil Nil = Nil
appendnil (Cons x xs) = main x xs
```

I will not be concerned with program extraction from process trees here.

11.1.1 Program inversion

My main interest in this chapter is to transform a *checker* into a description of the input that will satisfy the checker. That is, given a program that answers `true` or `false`, I will transform the program into a description of the input for which the checker answers `true`.

The above activity is generally known as *program inversion* when the description of the satisfying input is yet another program. It is, however, a nontrivial task to perform program inversion, as the following example shows.

Example 43 Consider a program that checks whether two lists are identical.

```
data List x = Nil | Cons x (List x)
main xs ys = same xs ys
same Nil ys = isnil ys
same (Cons x xs) ys = iscons ys x xs
isnil Nil = true
isnil ys = false
iscons (Cons y ys) x xs = if (x == y) (same xs ys) false
iscons ys x xs = false
if false x y = y
if true x y = x
```

The auxiliary functions `isnil` and `iscons` are needed because I only allow pattern matching on one argument at a time. The reason for having this restriction is that it associates every pattern match with a particular function definition.

The result of inverting the above program w.r.t. `true` should be another program that produces all pairs of identical lists. However, it is unreasonable to assume that I can produce such a program: Even though it is easy to imagine a program that produces an infinite stream of pairs of lists with identical spines, where should the elements come from? Based on their type, these elements could be enumerated, but such an enumeration clearly leads to non-termination in the general case. What is worse still, the imagined program will not per se give us a good *description* of the input set; I can merely get an indication of the input set by running it and observing its ever-increasing output. □

Instead of inverting a program, one might perform its computation backwards. A general method to perform *inverse computation* has been proposed by Abramov & Glück [1], namely the Universal Resolving Algorithm (URA). The URA constructs a process tree for the object program, and produces from the process tree a potentially infinite set of constraints on the uninstantiated input (variables xs and ys in the above example). Each constraint describes a *set* of input values by means of a *substitution/restriction* pair. The produced constraints are pairwise disjoint, in the sense that the sets they describe are pairwise disjoint. Variables can appear several times in each constraint, indicating identical values. For the above example, the URA would produce something like

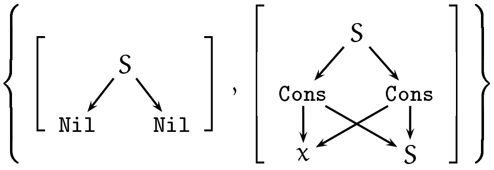
$$\begin{array}{l}([xs \mapsto \textsc{nil}, ys \mapsto \textsc{nil}], [])\\ ([xs \mapsto \textsc{cons}\ \mathcal{X}_1\ \textsc{nil}, ys \mapsto \textsc{cons}\ \mathcal{X}_1\ \textsc{nil}], [])\\ ([xs \mapsto \textsc{cons}\ \mathcal{X}_1\ (\textsc{cons}\ \mathcal{X}_2\ \textsc{nil}), ys \mapsto \textsc{cons}\ \mathcal{X}_1\ (\textsc{cons}\ \mathcal{X}_2\ \textsc{nil})], [])\\ \ldots\end{array} \tag{11.1}$$

Here the URA would never terminate. The merit of the URA is that it is sound and complete, so *if* it terminates, the result precisely captures the input set.

In this chapter I will sacrifice soundness to develop an approach that always produces a finite description of the satisfying input.

11.1.2 Overview

In Section 11.2, I present a formalisation of a certain kind of context-free grammars, namely *dag grammars*. For instance, the above checker can be described by the grammar



(11.2)

The grammar consists of two productions, each formed as an acyclic directed graph (also known as a *dag*). The first says that an S can be rewritten into a dag consisting of two single nodes labelled `Nil`. The second says that an S can be rewritten into a more complex dag with two roots. The two productions can be viewed as a finite representation of Equation 11.1. Such dag grammars can precisely express the data and control flow of a program, something which is not possible with standard tree grammars.

In Section 11.4, I present an automatic method to extract a dag grammar from a program. Conceptually, the extraction works in two steps: First I drive the object program to produce a process tree, second I extract a grammar from this process tree. A precursor for driving the program is a precise formulation of the semantics of the programming language, which I present in Section 11.3. The extracted dag grammars are approximations of the input set, and in Section 11.6, I consider various ways to improve the precision of my method.

As an application, I will in Section 11.5 show that, given a type checker and a λ-term, it is possible to derive a type scheme for the λ-term.

11.2 Dag grammars

Definition 77 Given a set S, a **graph** G over S consists of a label function $lab \in \mathbb{N} \Rightarrow S$ and a connected-by relation $\multimap \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, where $i \overset{k\;\ell}{\multimap} j$ means that node i's kth successor is node j and j's ℓth predecessor is i. The relation should satisfy the properties that there is a label to each node, and that the successors and predecessors are numbered consecutively from 0. Formally,

$$i \overset{k\;\ell}{\multimap} j \Rightarrow \left(\begin{array}{l} \{i, j\} \subseteq \mathscr{D}lab \\ \wedge\ \forall k' \in \{0, \ldots, (k-1)\}\ (\exists\{\ell', j'\} \subseteq \mathbb{N}\ (i \overset{k'\;\ell'}{\multimap} j')) \\ \wedge\ \forall \ell' \in \{0, \ldots, (\ell-1)\}\ (\exists\{k', i'\} \subseteq \mathbb{N}\ (i' \overset{k'\;\ell'}{\multimap} j)) \end{array} \right) .$$

When the order of successors and predecessors is immaterial, I simply use $\multimap$ as a binary relation and write $i \multimap j$ whenever $\exists\{k, \ell\} \subseteq \mathbb{N}\ (i \overset{k\ \ell}{\multimap} j)$. □

In the following, I will use subscripts like lab_G or $\underset{G}{\multimap}$ when it is otherwise not obvious which graph I refer to. By $\overset{+}{\multimap}$ I denote the transitive closure of $\multimap$. In general, I will superscript any binary relation with $+$ to denote its transitive closure, and I will superscript with $*$ to denote its reflexive closure.

Definition 78 A graph G is a **dag** when it contains no cycles (i.e., $\nexists i \in \mathbb{N}\ (i \overset{+}{\multimap} i)$). For dags, it is natural to talk of roots and leaves:

$$\begin{aligned} \mathrm{roots}_G &\overset{\mathrm{def}}{=} \{ i \in \mathbb{N} \mid \nexists j \in \mathbb{N}\ \ (j \underset{G}{\multimap} i) \} \\ \mathrm{leaves}_G &\overset{\mathrm{def}}{=} \{ i \in \mathbb{N} \mid \nexists j \in \mathbb{N}\ \ (i \underset{G}{\multimap} j) \} \end{aligned}$$

Two dags D and E are **equivalent**, denoted $D \sim E$, when D and E can be transformed into one another by renumbering the nodes, that is,

$$\exists \theta \in \mathbb{N} \leftrightarrow \mathbb{N}\ \ (\mathrm{lab}_D = \mathrm{lab}_E \circ \theta\ \wedge\ i \underset{D}{\overset{k\ \ell}{\multimap}} j \Leftrightarrow \theta i \underset{E}{\overset{k\ \ell}{\multimap}} \theta j)\ \ ,$$

where $S \leftrightarrow T$ denotes the set of bijections between S and T. The set of dags over S is denoted $\mathbb{D}S$. □

Example 44 Each of the two structures in Equation 11.2 depicts an *equivalence class* of dags over $\{S, \texttt{Nil}, \texttt{Cons}, x\}$ — in the sense that the structure describes a family of equivalent dags — because the node set ($\subset \mathbb{N}$) is left unspecified; the order of successors and predecessors, however, *is* specified by adopting the convention that heads and tails of arcs are ordered from left to right. A *concretisation* of the right dag is, for example,

$$\mathrm{lab} = \{0 \mapsto S\,,\ 1 \mapsto \texttt{Cons}\,,\ 2 \mapsto \texttt{Cons}\,,\ 3 \mapsto x\,,\ 4 \mapsto S\}$$

$$0 \overset{0\ 0}{\multimap} 1 \qquad 0 \overset{1\ 0}{\multimap} 2 \qquad 1 \overset{0\ 0}{\multimap} 3 \qquad 1 \overset{1\ 0}{\multimap} 4 \qquad 2 \overset{0\ 1}{\multimap} 3 \qquad 2 \overset{1\ 1}{\multimap} 4\ \ ,$$

and I have that leaves $\{3, 4\}$ and roots $\{0\}$. □

Definition 79 Given a set Σ, a **dag grammar** over Σ is a set of finite dags over Σ. The set of dag grammars over Σ is denoted $\mathbb{G}\Sigma$. □

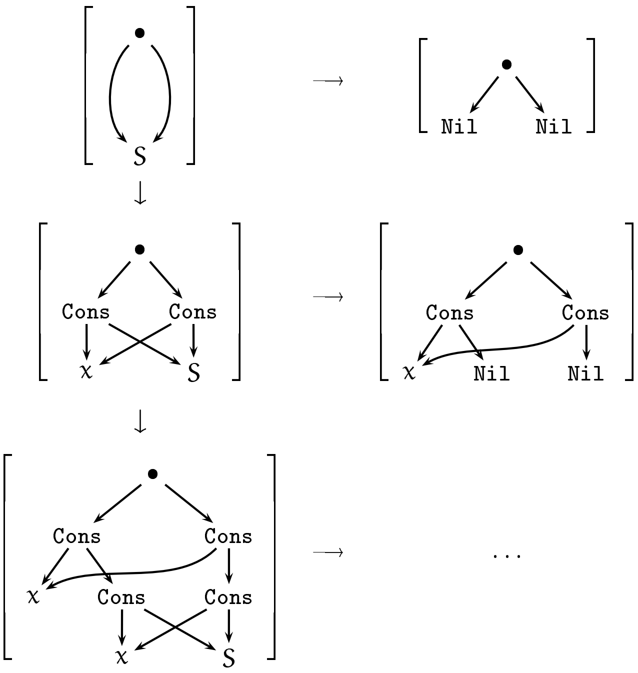
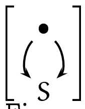


Figure 11.2: Dag rewrites

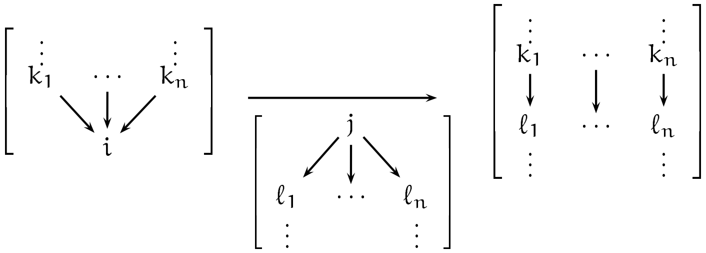
Example 45 The two dags in Equation 11.2 comprise a dag grammar over { S , x , Nil , Cons }. □

A dag grammar describes a *dag language*, in which every dag has been generated by a number of *rewrites* on an *initial* dag. Before I give the formal definition of rewrites and languages, an example is in order.

Example 46 The dag 

can be rewritten by the dag grammar in Equation 11.2 as depicted in Figure 11.2. The symbol • is here used to maintain an order between an otherwise unordered set of roots. □

Below you will find the formal definition of graph grammar rewrites. The example above hopefully illustrates how such rewrites work. Informally, a dag can be rewritten if it has a leaf node i that *matches* a root node j in a dag in the graph grammar. By *matches*, I mean that i and j have the same label, and that the number of predecessors of i matches the number of successors of j. The result of the rewrite is that the leaf i and root j dissolve, as it were, and the predecessors of i become the predecessors of the successors of j, in the right order, as illustrated below.



The following definitions generalise the above notion of rewriting to several leaves $i_0, \ldots, i_n$ and roots $j_0, \ldots, j_n$. I will need this generality in Section 11.4.

Definition 80 Given dags D and E, I define

$$\textit{match}\ D\ E \stackrel{\text{def}}{=} \left\{ \left\langle \begin{array}{c} \{i_0, \ldots, i_n\} \\ \{j_0, \ldots, j_n\} \end{array} \right\rangle \;\middle|\; \begin{array}{l} \forall m \in \{0, \ldots, n\} \\ \left(\begin{array}{l} (\mathsf{lab}_D\ i_m) = (\mathsf{lab}_E\ j_m) \\ \wedge\ i_m \in \mathsf{leaves}_D \\ \wedge\ j_m \in \mathsf{roots}_E \end{array} \right) \end{array} \right\}$$

and

$$\textit{maxmatch}\ D\ E \stackrel{\text{def}}{=} \left\{ \langle I, J \rangle \in \textit{match}\ D\ E \;\middle|\; \begin{array}{c} \forall \langle I', J' \rangle \in \textit{match}\ D\ E \\ (\ |I| \geq |I'| \wedge |I| > 0\ ) \end{array} \right\}$$

□

Definition 81 Given a relation $\rightarrow\ \subseteq S \times T$ and a set S', I define the **removal** of S' from $\rightarrow$ as $\rightarrow_{/S'} \stackrel{\text{def}}{=} \{\langle s, t \rangle \mid s \in (S \setminus S') \wedge s \rightarrow t\}$. The disjoint union of two binary relations is defined if their domains are disjoint. □

In the following, I carefully pay attention to the exact set of nodes ($\subset \mathbb{N}$) that each particular dag comprises. To avoid node clashes when I rewrite graphs, I use the fact that, given a finite dag G', there are infinitely many equivalent dags G.

Definition 82 Given a dag grammar $\Gamma \in \mathbb{G}\Sigma$, the **one-step-rewrite** relation $\underset{\Gamma}{\rightarrow} \subseteq \mathbb{D}\Sigma \times \mathbb{D}\Sigma$ is defined by

$$
D \underset{\Gamma}{\rightarrow} E \Leftrightarrow
\begin{array}{l}
\exists \{ G', G \} \subseteq \mathbb{D}\Sigma \\
\left(\begin{array}{l}
G' \in \Gamma \wedge G \sim G' \wedge \\
\exists \langle I, J \rangle \in \mathit{maxmatch}\ D\ G \\
\left(\begin{array}{l}
\mathrm{lab}_E = (\mathrm{lab}_D)_{/I} \uplus (\mathrm{lab}_G)_{/J} \ \wedge \\
\forall \{ i, j, k, \ell \} \subseteq \mathbb{N} \\
\left(\begin{array}{l}
i \overset{k\ \ell}{\underset{E}{\multimap}} j \Leftrightarrow \\
\qquad \left(\begin{array}{l}
\exists \{ m, c \} \subseteq \mathbb{N}\ (i \overset{k\ c}{\underset{D}{\multimap}} i_m \wedge j_m \overset{c\ \ell}{\underset{G}{\multimap}} j) \\
\vee\, i \overset{k\ \ell}{\underset{D}{\multimap}} j \wedge j \notin I \vee i \overset{k\ \ell}{\underset{G}{\multimap}} j \wedge i \notin J
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array}
$$

□

Definition 83 Given a dag grammar Γ and an *initial* dag I, the **dag language** $\mathscr{L}_I\Gamma$ is the set of normal forms: $\mathscr{L}_I\Gamma \overset{\text{def}}{=} \{ D \mid I \overset{*}{\underset{\Gamma}{\rightarrow}} D \wedge \nexists E\ (D \underset{\Gamma}{\rightarrow} E) \}$. □

I now have a grammar framework that can readily express sets of input. The next step is to show how a dag grammar can be extracted from a program. I start by defining the semantics of the object language, and, on top of the semantics, I then build the machinery that calculates the desired grammars.

11.3 Object language

My object language is a first-order functional language with pattern matching and operators for term equality and boolean AND.

Definition 84 Given a set of variables X, function names F, pattern-function names G, and constructors C (including the constants TRUE and FALSE), I define

$$\begin{array}{rrcll}
\mathit{Program} \ni & q & ::= & d_1 \ldots d_m & \text{(definitions)} \\
 & d & ::= & f\ x_1\ \ldots\ x_n = t & \text{(function)} \\
 & & \mid & g\ p_1\ x_1\ \ldots\ x_n = t_1 & \text{(matcher)} \\
 & & & \qquad\vdots & \\
 & & & g\ p_m\ x_1\ \ldots\ x_n = t_m & \\
 & p & ::= & x \mid c\ x_1\ \ldots\ x_n & \text{(pattern)} \\
\mathit{Term} \ni & t & ::= & x & \text{(variable)} \\
 & & \mid & t_1\ \&\ t_2 \mid t_1 \equiv t_2 & \text{(test)} \\
 & & \mid & c\ t_1\ \ldots\ t_n & \text{(construction)} \\
 & & \mid & f\ t_1\ \ldots\ t_n \mid g\ t_1\ \ldots\ t_m & \text{(application)}
\end{array}$$

where $n \geq 0$, $m > 0$, $x \in X$, $c \in C$, $f \in F$, and $g \in G$. As usual, I require that no variable occurs more than once in the left-hand side of a function definition (**left-linear**), and that all variables in the right-hand side of a function definition is a subset of those in the left-hand side (**closed**). Finally I require that the patterns defined for each g-function are pairwise distinct modulo variable names (**non-overlapping**); in particular, a g-function can have at most one "catch-all" pattern.

□

In concrete programs I use a Haskell-like syntax, including data-type definitions. The above syntax can be viewed as the intermediate language obtained after type checking, which rules out terms like NIL & NIL and NIL $\equiv$ TRUE.

Definition 85 Given a term t, I denote by $\mathscr{V}t \in X^{\star}$ the **variables of** t, collected from left to right. Term t is **ground** if $\mathscr{V}t = \langle\rangle$. □

Example 47 $\mathscr{V}(\text{TRIPLE}\ x\ y\ y) = \langle x\ ,\ y\rangle \neq \langle y\ ,\ x\rangle = \mathscr{V}(\text{TRIPLE}\ y\ y\ x)$. □

Definition 86 A function $\theta \in X \Rightarrow \mathit{Term}$ can be regarded as a **substitution** $\mathit{Term} \Rightarrow \mathit{Term}$ in the usual way, and, for $t \in \mathit{Term}$, I will write θt for the result of such a substitution. Given program q, writing, say, $g\ (c\ y_1\ \ldots\ y_m)\ x_1\ \ldots\ x_n \overset{q}{=} t$ means that in q there is a function definition identical to $g\ (c\ y_1\ \ldots\ y_m)\ x_1\ \ldots\ x_n = t$, up to variable naming. □

I will now give the normal-order semantics of the object language by defining a small-step relation that takes *redexes* into their *contracta*. I can separate redexes from their contexts by the following syntactic classes.

Definition 87

$$
\begin{array}{llccll}
\textbf{contexts} & D & \ni & d & ::= & e \mid c\ v_1\ \ldots\ v_n\ d\ t_1\ \ldots t_m \\
 & & & e & ::= & [\,] \mid g\ e\ t_1\ \ldots\ t_n \\
 & & & & \mid & e \equiv t \mid o \equiv e \mid e \,\&\, t \\
\textbf{redexes} & R & \ni & r & ::= & f\ t_1\ \ldots\ t_n \\
 & & & & \mid & g\ (c\ t_1\ \ldots\ t_m)\ t_{m+1}\ \ldots\ t_n \\
 & & & & \mid & c\ t_1\ \ldots\ t_n \equiv c'\ t'_1\ \ldots\ t'_m \\
 & & & & \mid & b \,\&\, t \\
\textbf{observables} & O & \ni & o & ::= & x \mid c\ t_1\ \ldots\ t_n \\
\textbf{values} & V & \ni & v & ::= & x \mid c\ v_1\ \ldots\ v_n \\
 & & & b & ::= & \textsc{false} \mid \textsc{true}\ .
\end{array}
$$

By $d[t]$ I denote the result of replacing $[\,]$ in d with term t. □

Any *ground* term t is either a value or can be uniquely decomposed into a context and a redex (i.e., $t = d[r]$). This decomposition property allows us to define the semantics as below. The semantics imposes left-to-right evaluation, except for the equality operator, which evaluates both its arguments until two outermost constructors can be compared.

Definition 88 Given a program q, the **small-step semantics** of q is defined by the smallest relation $\xrightarrow[\mathsf{inner}(q)]{} \subseteq R \times \mathit{Term}$ on closed terms as defined by Figure 11.3. □

To get the full semantics of the language, I simply need to close the $\xrightarrow[\mathsf{inner}(q)]{}$-relation under all contexts. The semantics is deterministic:

Lemma 41

$$\forall q \in \mathit{Program}, r \in R, \{\, t\,,\ t' \,\} \subseteq \mathit{Term}\ (\ r \xrightarrow[\mathsf{inner}(q)]{} t \wedge r \xrightarrow[\mathsf{inner}(q)]{} t' \Rightarrow t = t'\ )\quad .$$

PROOF (sketch). By induction and case analysis of the syntactic structure of terms: Each term either cannot be decomposed into a context and a redex, or it can be uniquely decomposed, in which case the redex has at most one small-step derivation. □

$$\begin{array}{l}
f\ t_1\ \ldots\ t_n \\
\qquad \xrightarrow[\text{inner}(q)]{} \{x_1 \mapsto t_1\,,\ \ldots\,,\ x_n \mapsto t_n\}t, \\
\qquad\qquad \text{if } f\ x_1\ \ldots\ x_n \overset{q}{=} t
\end{array}$$

$$\begin{array}{l}
g\ (c\ t_1\ \ldots\ t_m)\ t_{m+1}\ \ldots\ t_n \\
\qquad \xrightarrow[\text{inner}(q)]{} \{x_1 \mapsto t_1\,,\ \ldots\,,\ x_n \mapsto t_n\}t, \\
\qquad\qquad \text{if } g\ (c\ x_1\ \ldots\ x_m)\ x_{m+1}\ \ldots\ x_n \overset{q}{=} t
\end{array}$$

$$\begin{array}{l}
g\ (c\ t_{m+1}\ \ldots\ t_n)\ t_1\ \ldots\ t_m \\
\qquad \xrightarrow[\text{inner}(q)]{} \{x \mapsto c\ t_{m+1}\ \ldots\ t_n\,,\ x_1 \mapsto t_1\,,\ \ldots\,,\ x_m \mapsto t_m\}t, \\
\qquad\qquad \text{if } g\ x\ x_1\ \ldots\ x_m \overset{q}{=} t \wedge g\ (c\ \ldots)\ \ldots \overset{q}{\neq} \ldots
\end{array}$$

$$\begin{array}{l}
c\ t_1\ \ldots\ t_n \equiv c\ t'_1\ \ldots\ t'_n \\
\qquad \xrightarrow[\text{inner}(q)]{} \begin{cases} \text{TRUE}, & \text{if } n = 0 \\ t_1 \equiv t'_1\ \&\cdots\&\ t_n \equiv t'_n, & \text{if } n > 0 \end{cases}
\end{array}$$

$$c\ t_1\ \ldots\ t_n \equiv c'\ t'_1\ \ldots\ t'_m \xrightarrow[\text{inner}(q)]{} \text{FALSE}, \quad \text{if } c \neq c'$$

$$\text{FALSE}\ \&\ t \xrightarrow[\text{inner}(q)]{} \text{FALSE}$$

$$\text{TRUE}\ \&\ t \xrightarrow[\text{inner}(q)]{} t$$

Figure 11.3: Normal-order semantics

11.4 Explicitation

The previous section aloows us to deal with execution of ground terms. To be able to *drive* a program, however, I need to handle terms containing variables. I use the following syntactic class, in combination with the previously defined ones, to describe all terms that cannot be decomposed into contexts and redexes.

Definition 89

$$\begin{array}{lllll} \textbf{speculatives} & S & \ni & s & ::= & g\,x\,t_1\,\ldots\,t_n \\ & & & & | & o \equiv x \\ & & & & | & x \equiv o \\ & & & & | & x \,\&\, t \\ & & & & | & \textbf{let}\ x := t_1\ \textbf{in}\ t_2 \end{array}$$

As for the **variables** of a let-term, I let $\mathscr{V}(\textbf{let}\ x := t_1\ \textbf{in}\ t_2) \overset{\text{def}}{=} \mathscr{V} t_1 \cup (\mathscr{V} t_2 \backslash \{x\})$.

□

The use of let-terms will be described below. As in the previous section, I obtain a unique-decomposition property: *Any* term t is either a value, can be uniquely decomposed into a context and a redex, or can be uniquely decomposed into a context and a *speculative* (i.e., $t = d[s]$). The extra syntactic class s enables us to identify terms that need to be *driven* (i.e., instantiated).

In supercompilation, driving is used to obtain a process tree from the object program. The process tree serves two orthogonal purposes: It keeps track of data and control flow (essentially variable instantiations and recursive calls), and it provides a convenient way to monitor the driving process and perform the generalisations needed to ensure a finite process tree. When a generalisation is needed for a term t, t is replaced by a term "**let** $x := t_1$ **in** t_2" such that $t = t_2[x := t_1]$. The point of making such a generalisation is to be able to treat t_1 and t_2 independently. For an example, you might want to revisit Chapter 5.

In my approach to program inversion, called *explicitation*, I will assume that the generalisations necessary to ensure termination have been computed in advance by an *off-line generalisation analysis*. To be more specific, I assume that some terms have been replaced by terms of the form **let** $x := t_1$ **in** t_2 in the program of interest. With respect to the data and control flow of the program, the flow can be expressed by dag grammars, which I will elaborate on later in this section.

Since I have thus eliminated the need for a process tree, I will, to keep things simple, drive the object program without constructing the actual process tree. The construction of a process tree, although important in practice, is not the essence in my approach.

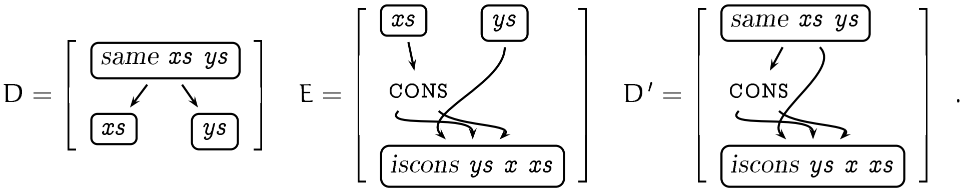
Remark 14 I should note here that the existence of an off-line generalisation analysis is not essential: The explicitation process described in the following could incorporate the non-termination detection and perform the necessary generalisations, as described in previous chapters. But because such an incorporation would induce unnecessary clutter in my presentation, I will concentrate on the description of how to extract a dag-grammar by driving.

From a bird's-eye view, the explicitation process of *a single branch of speculative execution* works as follows. Starting with the main term of the object program, a dag grammar Γ is gradually built up by driving the term: each time a speculative (cf. Definition 89) hinders normal reduction, I perform an instantiation to both the term and the dag grammar, such that reduction of the term can proceed and the grammar reflects the structure of the input variables. In fact, each driving step of a term results in a new production in the grammar, such that every term I meet while driving the program has its own production. When I meet a term which I have seen before, a loop is introduced into the grammar, and driving of the term stops. A term t that *cannot* be driven any further is compared to the output I desired, namely non-false output: If t is false, then the result is an empty grammar; otherwise it is the accumulated grammar Γ. In general, I parameterise the explicitation process over a *discrimination* function h. For my purpose, then,

$$h\ t\ \Gamma \stackrel{\text{def}}{=} \textbf{if}\ t = \text{FALSE}\ \textbf{then}\ \varnothing\ \textbf{else}\ \Gamma\ . \tag{11.3}$$

In the above fashion, I can produce a grammar for every speculative execution of the program: Each possible instantiation of a term gives rise to a slightly different term and grammar. The final grammar is then the union of the grammars produced for all executions.

As an example of an instantiation on a dag grammar, consider the dags



D represents a call to *same* where the arguments are unknown. E represents the body of *same*: a pattern match on the variable *xs* and a call to *iscons* with three arguments (cf. Example 43). The order of the arrows is important, since it defines the data flow. If I view $\{E\}$ as a dag grammar, D can be rewritten into D' by means of $\{E\}$, as shown above. Formally,

Definition 90 Given dags D and E, the **dag substitution** $\{E\}D$ is defined as (cf. Definition 80)

$$\{E\}D \overset{\text{def}}{=} \begin{cases} D', & \text{if } |\mathit{maxmatch}\ D\ E| = 1 \wedge D \xrightarrow[\{E\}]{} D' \\ D, & \text{otherwise.} \end{cases}$$

Substitutions are extended to grammars in the obvious way:

$$\{E\}\Gamma \overset{\text{def}}{=} \{D' \mid \exists D \in \Gamma\ (\{E\}D = D')\} \quad .$$

□

To construct dags from terms, I use the following shorthands.

Definition 91 Given a term t,

$$\top_t \overset{\text{def}}{=} \left[\begin{array}{c} t \\ \swarrow \quad \searrow \\ x_1 \cdots x_n \end{array}\right] \quad \text{and} \quad \bot_t \overset{\text{def}}{=} \left[\begin{array}{c} x_1 \cdots x_n \\ \searrow \quad \swarrow \\ t \end{array}\right] \quad ,$$

where $\mathscr{V}t = \langle x_1, \ldots, x_n \rangle$. □

The full explicitation process will be explained in detail below. Formally, it is summed up in the following definition.

$$\mathscr{E}_{\mathsf{h}}[\![q]\!] \;=\; \mathscr{E}_{\mathsf{h}}[\![t]\!]^{\varnothing}_{\varnothing} \text{ where } \mathit{main}\ \ldots \overset{q}{=} t$$

$$\mathscr{E}_{\mathsf{h}}[\![t]\!]^{\Gamma}_{\Omega} = \textbf{let } \Omega^{t} = \Omega \cup \{\, t \,\} \textbf{ in}$$

① $\text{if } t \in \Omega \text{ then } \Gamma$

② $\text{else if } \exists \theta \in (X \leftrightarrow X)\ (\,\theta t \in \Omega\,) \text{ then } \Gamma \cup \left\{ \begin{bmatrix} t \\ \overset{n}{\curvearrowright\cdots\curvearrowleft} \\ \theta t \end{bmatrix} \right\} \text{ where } n = |\mathscr{V} t|$

③ $\text{else if } t = d[r] \wedge r \xrightarrow[\mathsf{inner}(q)]{} t' \text{ then } \mathscr{E}_{\mathsf{h}}[\![d[t']]\!]^{\Gamma'}_{\Omega^{t}},$

$\quad\text{where } \Gamma' = \Gamma \cup \big(\big\{\, \bot_{d[t']} \,\big\}\, \top_{t}\big)$

④ $\text{else if } t = d[\textbf{let } x := t_1 \textbf{ in } t_2] \text{ then } \mathscr{E}_{\lambda x y . y}[\![t_1]\!]^{\Gamma'}_{\Omega^{t}} \cup \mathscr{E}_{\mathsf{h}}[\![t']\!]^{\Gamma'}_{\Omega^{t}}$

$\qquad\text{where } \quad t' = \{\, x \mapsto y \,\}\, d[t_2],\ y \notin \mathscr{V} d[t_2]$

$\qquad\quad\text{and } \quad \Gamma' = \Gamma \cup (\{\, \bot_{t_1} \,\}(\{\, \bot_{t'} \,\}\, \top_{t}))$

⑤ $\text{else if } t = d[g\ x\ t_1\ \ldots\ t_n] \text{ then } \bigcup_{g\ (c\ x_1\ \ldots\ x_m)\ y_1\ \ldots\ y_n \overset{q}{=} u} \mathscr{E}_{\mathsf{h}}[\![t']\!]^{\Gamma'}_{\Omega^{t}}$

$\qquad\text{where } \quad t' = \{\, x \mapsto c\ x_1\ \ldots\ x_m \,\}(d[\{\, y_1 \mapsto t_1\,,\ \ldots\,,\ y_n \mapsto t_n \,\}u])$

$\qquad\quad\text{and } \quad \Gamma' = \Gamma \cup \left(\{\, \bot_{t'} \,\} \left(\left\{ \begin{bmatrix} & x & \\ & \downarrow & \\ & c & \\ \swarrow & & \searrow \\ x_1 & \cdots & x_m \end{bmatrix} \right\} \top_{t} \right) \right)$

(*Continues in Figure 11.5 . . .*)

Figure 11.4: Explicitation by driving

⑥ else if $t = d[c\ t_1\ \ldots\ t_n \equiv x]$ then $\mathscr{E}_h[\![d[x \equiv c\ t_1\ \ldots\ t_n]]\!]^{\Gamma}_{\Omega}$

⑦ else if $t = d[x \equiv y] \wedge y \in X$ then $\mathscr{E}_h[\![t']\!]^{\Gamma'}_{\Omega^t} \cup \mathscr{E}_h[\![t'']\!]^{\Gamma''}_{\Omega^t}$

where $t' = \{x \mapsto y\}(d[\textsc{true}])$,

and $$\Gamma' = \Gamma \cup \Bigg(\{\bot_{t'}\} \Bigg(\Bigg(\Bigg\{ \left[\begin{tikzcd} x \arrow[dr] & & y \arrow[dl] \\ & \nabla \arrow[d] & \\ & y & \end{tikzcd} \right] \Bigg\} \top_t \Bigg) \Bigg)$$

and $t'' = d[\textsc{false}]$, $\Gamma'' = \Gamma \cup (\{\bot_{t''}\}\top_t)$

⑧ else if $t = d[x \equiv c\ t_1\ \ldots\ t_n]$ then $\mathscr{E}_h[\![t']\!]^{\Gamma'}_{\Omega^t} \cup \mathscr{E}_h[\![t'']\!]^{\Gamma''}_{\Omega^t}$

where $\theta = \{x \mapsto c\ x_1\ \ldots\ x_n\}, x_1, \ldots, x_n \notin \mathscr{V}t$,

and $t' = \theta(d[\textsc{true}\ \&\ x_1 \equiv t_1\ \&\ \cdots\ \&\ x_n \equiv t_n])$,

and $$\Gamma' = \Gamma \cup \Bigg(\{\bot_{t'}\} \Bigg(\Bigg(\Bigg\{ \left[\begin{tikzcd} & x \arrow[d] & \\ & c \arrow[dl] \arrow[dr] & \\ x_1 & \cdots & x_n \end{tikzcd} \right] \Bigg\} \top_t \Bigg) \Bigg)$$

and $t'' = d[\textsc{false}]$, $\Gamma'' = \Gamma \cup (\{\bot_{t''}\}\top_t)$

⑨ else if $t = d[x\ \&\ u]$ then $\mathscr{E}_h[\![t'']\!]^{\Gamma''}_{\Omega^t} \cup \mathscr{E}_h[\![t']\!]^{\Gamma'}_{\Omega^t}$

where $t' = \{x \mapsto \textsc{true}\}\, d[u]$,

and $$\Gamma' = \Gamma \cup \Bigg(\{\bot_{t'}\} \Bigg(\Bigg(\Bigg\{ \left[\begin{tikzcd} x \arrow[d] \\ \mathrm{TRUE} \end{tikzcd} \right] \Bigg\} \top_t \Bigg) \Bigg)$$

and $t'' = \{x \mapsto \textsc{false}\}\, d[\textsc{false}]$,

and $$\Gamma'' = \Gamma \cup \Bigg(\{\bot_{t''}\} \Bigg\{ \left[\begin{tikzcd} x \arrow[d] \\ \mathrm{FALSE} \end{tikzcd} \right] \Bigg\} \top_t \Bigg)$$

⑩ else $h\ t\ (\Gamma \cup \top_t)$

Figure 11.5: Explicitation by driving (cont.)

Definition 92 Given a program q and a function $\mathrm{h} \in \mathit{Term} \Rightarrow \mathbb{G}(\mathit{Term} \cup \mathrm{C}) \Rightarrow \mathbb{G}(\mathit{Term} \cup \mathrm{C})$, the **explicitation** of q is a dag grammar $\mathscr{E}_{\mathrm{h}}[\![\mathrm{q}]\!]$, as described in Figure 11.4. □

The following explanation of the explicitation process carries a number of references that hopefully will guide the understanding of Figure 11.4.

Explicitation starts with the main term t of the program, an empty dag grammar Γ, and an empty set of previously seen terms Ω. In each step, I inspect the structure of the current term t, and either stop, or add a production for t to Γ and make a new step with t added to the seen-before set.

① If t has been seen before, a production for t is already present in the grammar Γ, so I return the accumulated Γ unchanged. ② If a renaming of t has been seen before (captured by a bijection θ), I introduce recursion in the grammar by adding a production that connects t to the (previously seen) $\theta\mathrm{t}$, respecting the number of variables.

③ If a redex can be unfolded using the standard semantics (cf. Definitions 87 & 88), a production linking t to its unfolding is added to Γ, and the process continues with the unfolding.

④ If a generalised term hinders unfolding, that is $\mathrm{t} = \mathrm{d}[\mathbf{let}\ \mathrm{x} := \mathrm{t}_1\ \mathbf{in}\ \mathrm{t}_2]$, $\mathrm{d}[\mathrm{t}_2]$ and t_1 are processed independently. Therefore, a production is added to the grammar such that t is linked to both $\mathrm{d}[\mathrm{t}_2]$ and t_1. This production will have some dangling roots[1] (namely x and $\mathscr{V}\mathrm{t}_1 \cap \mathscr{V}\mathrm{t}_2$) which reflect that the data flow is approximated. Because the traces of t_1 will tell us nothing about the output of t, the function h (cf. Equation 11.3) that is supposed to discriminate between various output is replaced by the function $\lambda \mathrm{x}\mathrm{y}.\mathrm{y}$ which does not discriminate: It always returns the accumulated grammar.

⑤ For a pattern matching function, the process is continued for all defined patterns. For each pattern p, I substitute the arguments into the matching body, and put it back into the context, which in turn receives the instantiation $\{\mathrm{x} \mapsto \mathrm{p}\}$, and I add to the grammar a production reflecting this instantiation.

For comparisons, there are three cases. ⑥ The first simply makes sure that variables are on the left side of the comparison. That settled, ⑦ if the right-hand side is another variable, two possibilities are explored: Either the comparison will fail, and hence I replace the speculative with FALSE; or the comparison will succeed, and I replace the speculative with TRUE and update the grammar and

[1] Unmatched roots are allowed in dag rewrites, cf. Definition 82.

$$
\begin{array}{l}
\mathscr{E}_{\mathsf{h}}\llbracket \mathit{same}\ \mathit{xs}\ \mathit{ys}\rrbracket^{\varnothing}_{\varnothing} \\
= \bigcup \left\{ \begin{array}{l}
\mathscr{E}_{\mathsf{h}}\llbracket \mathit{isnil}\ \mathit{ys}\rrbracket^{\{\,\mathsf{A}\,\}}_{\{\,\mathit{same}\ \mathit{xs}\ \mathit{ys}\,\}} = \bigcup \left\{ \begin{array}{l}
\mathscr{E}_{\mathsf{h}}\llbracket \mathtt{TRUE}\rrbracket^{\{\,\mathsf{A}\,,\,\mathsf{B}\,\}}_{\{\,\mathit{same}\ \mathit{xs}\ \mathit{ys}\,,\,\mathit{isnil}\ \mathit{ys}\,\}} = \{\,\mathsf{A}\,,\,\mathsf{B}\,\} \\
\mathscr{E}_{\mathsf{h}}\llbracket \mathtt{FALSE}\rrbracket^{\{\,\mathsf{A}\,,\,\mathsf{C}\,\}}_{\{\,\mathit{same}\ \mathit{xs}\ \mathit{ys}\,,\,\mathit{isnil}\ \mathit{ys}\,\}} = \varnothing
\end{array} \right. \\
\mathscr{E}_{\mathsf{h}}\llbracket \mathit{iscons}\ \mathit{ys}\ \mathit{x}\ \mathit{xs}\rrbracket^{\{\,\mathsf{D}\,\}}_{\{\,\mathit{same}\ \mathit{xs}\ \mathit{ys}\,\}} \\
\quad = \bigcup \left\{ \begin{array}{l}
\mathscr{E}_{\mathsf{h}}\llbracket \mathtt{FALSE}\rrbracket^{\{\,\mathsf{D}\,,\,\mathsf{E}\,\}}_{\{\,\mathit{same}\ \mathit{xs}\ \mathit{ys}\,,\,\mathit{iscons}\ \mathit{ys}\ \mathit{x}\ \mathit{xs}\,\}} = \varnothing \\
\mathscr{E}_{\mathsf{h}}\llbracket \mathit{if}\ (x \equiv y)\ \mathit{same}\ \mathit{xs}\ \mathit{ys}\ \mathtt{FALSE}\rrbracket^{\{\,\mathsf{D}\,,\,\mathsf{F}\,\}}_{\{\,\mathit{same}\ \mathit{xs}\ \mathit{ys}\,,\,\mathit{iscons}\ \mathit{ys}\ \mathit{x}\ \mathit{xs}\,\}} \\
\quad = \bigcup \left\{ \begin{array}{l}
\mathscr{E}_{\mathsf{h}}\llbracket \mathit{if}\ \mathtt{FALSE}\ (\mathit{same}\ \mathit{xs}\ \mathit{ys})\ \mathtt{FALSE}\rrbracket^{\{\,\mathsf{D}\,,\,\mathsf{F}\,,\,\mathsf{G}\,\}}_{\{\,\ldots\,,\,\mathit{if}\ (x\equiv y)\ (\mathit{same}\ \mathit{xs}\ \mathit{ys})\ \mathtt{FALSE}\,\}} \\
= \mathscr{E}_{\mathsf{h}}\llbracket \mathtt{FALSE}\rrbracket^{\{\,\mathsf{D}\,,\,\mathsf{F}\,,\,\mathsf{G}\,,\,\mathsf{H}\,\}}_{\{\,\ldots\,,\,\mathit{if}\ (x\equiv y)\ (\mathit{same}\ \mathit{xs}\ \mathit{ys})\ \mathtt{FALSE}\,,\,\mathit{if}\ \mathtt{FALSE}\ (\mathit{same}\ \mathit{xs}\ \mathit{ys})\ \,\}} \\
= \varnothing \\
\mathscr{E}_{\mathsf{h}}\llbracket \mathit{if}\ \mathtt{TRUE}\ (\mathit{same}\ \mathit{xs}\ \mathit{ys})\ \mathtt{FALSE}\rrbracket^{\{\,\mathsf{D}\,,\,\mathsf{F}\,,\,\mathsf{I}\,\}}_{\{\,\ldots\,,\,\mathit{if}\ (x\equiv y)\ (\mathit{same}\ \mathit{xs}\ \mathit{ys})\ \mathtt{FALSE}\,\}} \\
= \mathscr{E}_{\mathsf{h}}\llbracket \mathit{same}\ \mathit{xs}\ \mathit{ys}\rrbracket^{\{\,\mathsf{D}\,,\,\mathsf{F}\,,\,\mathsf{I}\,,\,\mathsf{J}\,\}}_{\{\,\mathit{same}\ \mathit{xs}\ \mathit{ys}\,,\,\ldots\,\}} \\
= \{\,\mathsf{D}\,,\,\mathsf{F}\,,\,\mathsf{I}\,,\,\mathsf{J}\,\}
\end{array} \right.
\end{array} \right.
\end{array} \right.
\end{array}
$$

Figure 11.6: Explicitation of the *same*-program

the context to reflect that both sides must be identical. In the grammar, the equal variables are coalesced by means of a special symbol ∇, which is needed to maintain the invariant that the in/out degree of terms correspond to the number of distinct variables. ⑧ If the right-hand side is an n-ary constructor, either the comparison will fail (as above), or it will succeed, in which case I will propagate that the variable must have a particular outermost constructor, of which the children must be tested for equality with the children of the constructor.

⑨ If a boolean expression depends on a variable, then the variable will evaluate to either TRUE or FALSE, and this information is propagated to each branch.

⑩ Terms that *cannot* be driven any further (i.e., $\mathsf{t} \in \mathcal{V}$) are fed to the function h, which in turn decides what to do with the accumulated grammar.

Example 48 Let q be the program from Example 43 and h defined as Equation 11.3. The explicitation $\mathscr{E}_{\mathsf{h}}\llbracket \mathsf{q}\rrbracket$ is depicted in Figure 11.6. The grammar produced is $\{\,\mathsf{A}\,,\,\mathsf{B}\,,\,\mathsf{D}\,,\,\mathsf{F}\,,\,\mathsf{I}\,,\,\mathsf{J}\,\}$. □

The derived grammars are sub-optimal: Most of the productions are intermediate, in the sense that they can be directly unfolded, or in-lined as it were. I

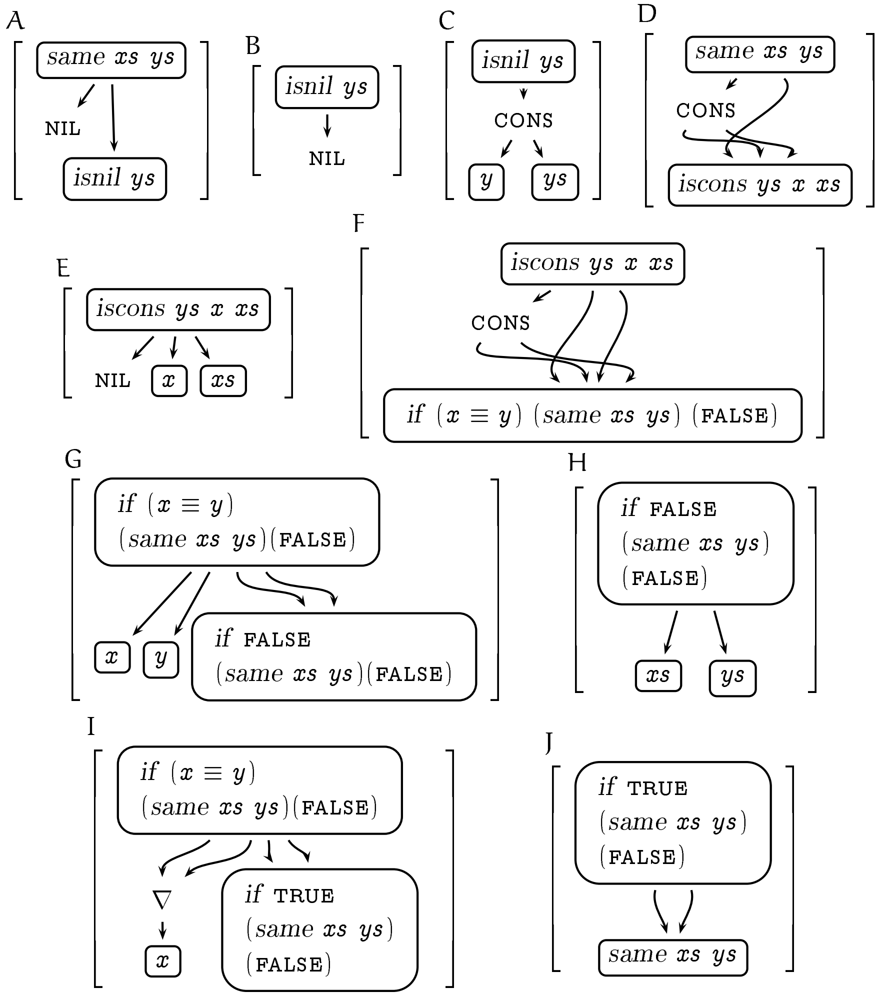


Figure 11.7: Explicitation of the *same*-program (cont.)

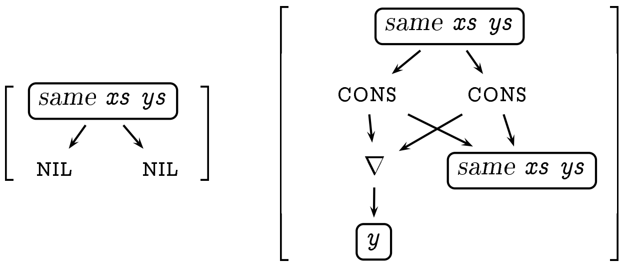
say that a grammar is *normalised* if it contains no intermediate productions, and I can easily normalise a grammar.

Definition 93 A dag grammar Γ can be **normalised**, denoted $\widehat{\Gamma}$, as follows.

$$\widehat{\Gamma} = \begin{cases} \{\,G\,\}\widehat{(\Gamma_{/G})}, & \text{if } \exists G \in \Gamma \left(\begin{array}{ll} & \mathrm{roots}_G \lightning \mathrm{leaves}_G \\ \wedge & \forall H \in \Gamma_{/G} \; (\, \mathrm{roots}_H \lightning \mathrm{roots}_G \,) \\ \wedge & \exists H \in \Gamma_{/G} \; (\{\,G\,\}H \nsim H\,) \end{array} \right) \\ \Gamma, & \text{otherwise.} \end{cases}$$

□

Example 49 If I normalise the grammar from Example 48, I almost get the grammar I promised in the introduction:



In this particular grammar, the bookkeeping symbol ∇ can be eliminated by yet another normalisation process, if so desired. □

Given that the object program contains the necessary generalisations, the lemma below is not surprising. However, if I incorporated a standard *on-line* termination strategy into the explication process, as explained in Remark 14, the following lemma would still hold.

Lemma 42 *Any explicitation produces a finite grammar.*

PROOF. The process tree of the program is finite (since the program contains the necessary generalisations), and thus a finite number of dags will be produced, assuming the h function is to be total and computable. □

More interestingly, the explicitation returns a dag grammar that contains all solutions. To express such a completeness theorem, I need a precise formulation of the set of terms generated by a dag grammar, as given below. Informally, a term is extracted from a dag simply by following the edges from left to right, collecting the labels — except when the label is a variable or the bookkeeping symbol ∇: Every variable is given a distinct name, and ∇ is treated as an indirection and left out in the term.

Definition 94 Given a dag grammar Γ, a label S with *arity* n, and a set of variables $X = \{x_0, x_1, \ldots\}$, the **term language** $\mathscr{T}_S^n\Gamma$ is the set of tuples of terms that can be extracted from the underlying dag language:

$$\mathscr{T}_S^n\Gamma \overset{\text{def}}{=} \left\{ \langle t_1, \ldots, t_n \rangle \;\middle|\; I = \begin{bmatrix} 0{:}\,\bullet \\ \overset{n}{\cdots} \\ S \end{bmatrix} \begin{array}{l} \wedge\, D \in \mathscr{L}_I\Gamma \\ \wedge\, \langle\!| D |\!\rangle_0 = \bullet\ t_1\ \ldots\ t_n \end{array} \right\}$$

$$\langle\!| D |\!\rangle_i \overset{\text{def}}{=} \begin{cases} x_i, & \text{if } L \in X \\ \langle\!| D |\!\rangle_j, & \text{if } L = \nabla \wedge i \multimap j \\ L\ \langle\!| D |\!\rangle_{j_0}\ \ldots\ \langle\!| D |\!\rangle_{j_m}, & \text{otherwise.} \end{cases}$$
$$\begin{array}{rl} \text{where} & L = \mathrm{lab}_D i \\ \text{and} & \{\langle i, 0, \ell_0, j_0\rangle, \ldots, \langle i, m, \ell_m, j_m\rangle\} \\ & \quad = \{\langle i, k, \ell, j\rangle \mid \exists\{k, \ell, j\} \subseteq \mathbb{N}\ (i \overset{k\ \ell}{\multimap} j)\} \end{array}$$

□

I can now relate all possible executions of the program to the set of terms generated by its explicitation.

Theorem 5 *Given a program* q *where main* $\ldots \overset{q}{=} t$ *and* $\mathscr{V}t = \langle x_1, \ldots, x_n\rangle$, *it holds that*

$$\begin{array}{l} \forall\{v_1, \ldots, v_n\} \subseteq V \\ \left(\begin{array}{c} t[x_1 := v_1, \ldots, x_n := v_n] \xrightarrow[\mathsf{inner}(q)]{*} v \in (V \setminus \{\textbf{FALSE}\}) \\ \Rightarrow \\ \exists\langle v_1', \ldots, v_n'\rangle \in \mathscr{T}_t^n(\mathscr{E}_h[\![q]\!]),\ \theta \in X \Rightarrow V \\ (\langle v_1, \ldots, v_n\rangle = \langle v_1'\theta, \ldots, v_n'\theta\rangle) \end{array} \right) \end{array} .$$

11.5 Application: Type inference

I now show that a type checking algorithm can be transformed into a type inference algorithm by explicitation. Specifically, I check that a λ-calculus expression has a given type in a given environment, using the standard relation

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \uplus \{x \mapsto \tau\} \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \to \sigma} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \quad .$$

As an example, consider the expression

$$\varnothing \vdash \lambda x.\lambda y.\lambda z.(xz)(yz) : \; ? \;,$$

meaning "what is the type of $\lambda x.\lambda y.\lambda z.(xz)(yz)$ in the empty environment?". I would expect the answer to be something like

$$\forall \alpha\beta\gamma \, [(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma] \quad . \tag{11.4}$$

I will now perform explicitation of the above expression. The type checker shown in Figure 11.8 takes an encoding of a proof tree P, an expression M, and an environment Γ, and returns `true` if P is indeed a valid proof of the type of M in Γ. In this encoding,[2] $\lambda x.\lambda y.\lambda z.(xz)(yz)$ becomes

```
S = Abs 0 (Abs 1 (Abs 2 (App (App (Var 0) (Var 2)) (App (Var 1) (Var 2)))))
```

and $\varnothing$ becomes `Empty`. If I want to explicitate the above expression, I can add the definition

```
main typ prf = typechk (Infer prf typ) S Empty
```

to the implementation of the type checker in Figure 11.8, which I then refer to as the specialised typechecker. Explicitation of the specialised typechecker gives a term language that consists of a single pair

```
 <(Ar (Ar x (Ar y z)) (Ar (Ar x y) (Ar x z))) , (...)>
```

The first element is indeed the encoding of type in Equation 11.4, and the second is the proof tree, which I have left out.

[2] The implementation assumes a primitive integer type `Int`.

```
data Type = Tv Int | Ar Type Type
data Term = Var Int | App Term Term | Abs Int Term
data Env = Empty | Bind Int Type Env
data Proof = Infer Premise Type
data Premise = Axiom | Elim Proof Proof | Intro Proof
typechk (Infer x y) z v = expchk z x y v
expchk (Var x) y z v = axiomchk y x z v
expchk (App x y) z v w = elimchk z x y v w
expchk (Abs x y) z v w = introchk z x y v w
axiomchk Axiom x y z = match z x y
axiomchk x y z v = false
elimchk (Elim x y) z v w x1 = typechk x z x1 & typechk y v x1 &
                              conclusion x == Ar (conclusion y) w
elimchk x y z v w = false
introchk (Intro x) y z v w = arrowcheck v x y z w
introchk x y z v w = false
match (Bind x y z) v w = if (v == x) (w == y) (match z v w)
match x y z = false
arrowcheck (Ar x y) z v w x1 = typechk z w (Bind v x x1) &
                               y == conclusion z
arrowcheck x y z v w = false
conclusion (Infer x y) = y
if false x y = y
if true x y = x
```

Figure 11.8: Type checker.

11.6 Improving soundness

Any automatic method for inversion of Turing-complete programs will have to make a compromise with respect to completeness, soundness and termination. I have made a compromise that will result in possibly unsound results: The explicitation of a program can produce a grammar that generates too many terms. From a practical point of view, I feel that this is the right compromise: It is better to obtain an approximation than not obtaining an answer at all. Moreover, explicitation can produce grammars that precisely identifies the input set, as seen from the two examples is previous sections, which indicates that explicitation is tight enough for at least some practical problems.

However, it still remains to identify exactly what causes loss of soundness in the general case. My method is inherently imprecise for three reasons.

Generalisations cause terms to be split up in separate parts (by means of let-terms). This prevents instantiations in the left branch of the process tree to affect the right branch, and vice versa.

Negative information is not taken in to account while driving. For example, driving the speculative term $x \equiv y$ will not propagate the fact that $x \neq y$ to the right branch of the process tree (although the fact that $x = y$ *is* propagated to the left branch, by means of a substitution). Moreover, the dag grammars cannot represent negative information.

Occur check is not performed on speculative terms like $x \equiv c\ x_1\ \ldots\ x_n$, that is, a situations where $x = x_i$ for some $i \leq n$ is not discovered. Obviously, such an equality would never imply that $x = c\ x_1\ \ldots\ x_n$. Similarly, the symmetric property that $x \equiv x$ never implies $x \neq x$ is not used either.

The occur check and its counterpart can easily be incorporated into rules ⑧ and ⑦, respectively, in Figure 11.4. Interestingly, explicitation of the type checker (specialised to a λ-term) would be sound, had these checks been incorporated.

As for negative information, I have described how to handle and propagate such information in another paper [97]. Incorporating negative information as proposed in that paper would be a simple task. Incorporating negative information into the dag grammars, however, would destroy their simplicity and thus severely cripple their usability.

Hence, generalisation and the inability of dag grammars to represent negative information are the true culprits. One could therefore imagine a variant of the explicitation algorithm — let's call it EXP — where one has incorporated the extensions I have suggested above. To improve soundness of EXP, one should target the way generalisations are carried out.

I am now in a position to conjecture the following roundabout relationship between EXP and the URA [1] described in the introduction: Given a program q without generalisations, EXP terminates on q whenever URA terminates on q. Moreover, if the result of URA contains no restrictions (negative information), the resulting grammar of EXP is sound.

11.7 Related work

Program inversion

In the literature, most program inversion is carried out by hand. One exception is Romanenko [89], who describes a pseudo algorithm for inverting a small class of programs written in Refal. In a later paper, Romanenko [90] extends the Refal language with non-deterministic construct, akin to those seen in logic programming, to facilitate exhaustive search. He also considers inverting a program with respect to a subset of its parameters, so-called *partial inversion*. I would like to extend my method to handle partial inversion, but as of this writing it is unclear how this should be done.

The only fully automated program inversion method I know of, is the one presented by Abramov & Glück [1]. Their method constructs a process tree from the object program, and solutions are extracted from the leaves of the tree in form of substitution/restriction pairs. The process trees are *perfect* (Glück & Klimov [37]) in the sense that no information is lost in any branch (completeness), and every branching node divides the information in disjoint sets (soundness). Unfortunately, soundness together with completeness implies non-termination for all but a small class of programs. The method I have presented here sacrifices soundness for termination.

What is common to both my and the above methods is that they all build upon the ground breaking work of Turchin and co-workers. The first English paper that contains examples of program inversion by driving seems to be [114].

For more references, see Sørensen & Glück [38].

Grammars

The idea of approximating functional programs by grammars is not new. For instance, Jones [46] presents a flow analysis for non-strict languages by means of tree grammars. Based on this work, Sørensen has developed a technique using tree grammars to approximate termination behaviour of deforestation [105]. Tree grammars, however, cannot capture the precise data and control flow: By definition, branches in a tree grammar are developed independent, which renders impossible the kind of synchronisation I need between variable and function calls. The dag grammars I have presented, precisely capture the data and control flow for any single speculative computation trace; synchronisation can only be lost when several alternative recursive productions exist in the grammar.

It seems possible to devise a more precise flow analysis based on dag grammars, which could be used along the lines of [105]. Indeed, the way I use dag grammars to trace data and control flow, turns out to have striking similarities with the use of *size-change graphs*, presented by Lee, Jones & Ben-Amram [63]. Size-change graphs approximate the data flow from one function to another, by capturing size-changes of parameters.

The dag-rewrite mechanism I have presented turns out to have a lot in common with the *fan-in/fan-out* rewrites presented by Lamping [60], in the quest for optimal reduction in the λ-calculus. The fan-in/fan-outs represent a complex way of synchronising different parts of a λ-graph, whereas my dag rewrites only perform a simple use-once synchronisation. The rewriting mechanism is also akin to *graph substitution* in hyper-graph grammars (see Bauderon & Courcelle [5] for a non-category-theory formulation), except that I allow any number of leaves to be rewritten and do not allow inner nodes to be rewritten. Strength-wise, hyper-graph grammars are apparently equivalent to attribute grammars. At present, I am not sure of the generative power of dag grammars.

11.8 Conclusion and future work

I have developed a method for inverting a given program with respect to a given output. The result of the inversion is a finite dag grammar that gives a complete description of the input: “Running” the dag grammar produces a (possibly infinite) set of terms that will contain all tuples of terms that result in the given output.

The method seems to be particularly useful when the object program is a checker, that is, one that returns either true or false. I have exemplified this by deriving a type scheme from a type checker and a given λ-term, thus effectively synthesising a type inference algorithm. Following this line, one could imagine a program that checks whether a document is valid XML. Inverting this program would result in a dag grammar, which could then be compared to a specification for valid XML, as a means of verifying the program. Inverting the program when specialised to a particular document, would result in a Document Type Definition. One could even imagine inverting a proof-carrying-code verifier [74] with respect to a particular program, thus obtaining a proof skeleton for the correctness of the code.

Further experiments with the above kinds of applications should be carried out to establish the strength and usability of my method.

Chapter 12

Interactive Configuration

A product line is a set of products and features with constraints on which subsets are available. For example, a company may sell a number of different computers (lap-tops, desk-tops, etc.) with different storage device, memory size, etc., but some combinations may be unavailable (e.g., floppy drive and CD-ROM may be mutually exclusive in lap-top models).

It is natural for the vendor to make a system available to the consumer in which he can experiment with the different options, for instance, how the selection of one product or feature entails or precludes the selection of another product or feature. Indeed, many such systems have appeared in online-shops on the internet. I will collectively refer to such system as *configurators*.

In most existing configurators, it is not possible to deselect an entity; selection can only be back-tracked, but there is no way to deselect an entity selected several clicks ago without also undoing all the intermediate selections. In this chapter, I will describe how deselections can be treated analogous to selections, making them first-class citizens of the approach.

The main advantage of the approach is not the derivation of any new efficient constraint algorithms, but rather some insights concerning what type of constraint propagation might be desirable in some configurators and concerning how a user might interact with such constraint propagation algorithms.

There are models significantly more complex than what I shall describe here. For example, a customer might wants to purchase a house as close to city centre as possible for under £20000. A *qualitative* constraint like this can only be solved by fully-fledged *constraint solvers* (e.g., Berkeley ANalysis Engine[1]). However, the working hypothesis in this chapter is that there are cases, particularly in online shopping, where more complex models are neither required nor desired.

12.1 Configurations and constraints

What we buy from a product line are *entities*. For instance, when we buy a car, the entities are the size of the engine, the number of doors, the type of transmission, etc. In a computer, they are the different hardware components.

We configure the product we are buying by selecting and deselecting entities. For instance, when we buy a computer, we select or deselect the floppy-drive entity, and we select or deselect the CD-ROM entity. A specification, for each entity (whether it is selected, deselected, or unspecified), will be called a configuration. A configuration in which every entity is either selected or deselected (and hence not unspecified) is called *total*.

Definition 95 Given a set of entities *Entity*, I define the following sets:

$$\begin{array}{rcll} \textit{Configuration} & \ni & a & ::= \{ e_1 \mapsto s_1 \, , \, \ldots \, , \, e_n \mapsto s_n \} \\ \textit{Specification} & \ni & s & ::= \textbf{selected} \\ & & & \;\;\,\mid \;\; \textbf{deselected} \\ & & & \;\;\,\mid \;\; \textbf{unspecified} \end{array}$$

where $e \in \textit{Entity}$. A configuration a is *total* iff $\forall e \in \mathscr{D} a \; (\, a e \neq \textbf{unspecified} \,)$. Conversely, a is said to *empty* iff $\forall e \in \mathscr{D} a \; (\, a e = \textbf{unspecified} \,)$. □

When we select from a product line, we cannot choose whatever we like: Our selection is constrained by the vendor. The constraints may state such properties as mutual exclusion of selection of two entities, implication from selection of one entity to another, etc. In general, any logical relation between selection of any number of entities is possible, but it is well-known (see e.g., Mendelson [69]) that all such logical relations can be expressed in terms of implication and negation.

[1] `http://www.cs.berkeley.edu/Research/Aiken/bane.html` .

Definition 96 Given a set of entities *Entity*, I define the set of constraints by

$$\begin{array}{rclll} \mathit{Constraint} & \ni & \mathrm{c} & ::= & \mathrm{e} & \text{(entity)} \\ & & & | & \mathrm{c} \Rightarrow \mathrm{c}' & \text{(implication)} \\ & & & | & \neg\mathrm{c} & \text{(negation)} \end{array}$$

where $\mathrm{e} \in \mathit{Entity}$. I define $\mathscr{D}\mathrm{c}$ to be the set of entities mentioned in c. □

Remark 15 In the following, I will use other connectives (conjunction, disjunction, etc.) in constraints and implicitly assume that such connectives are translated into appropriate constellations of implications and negations. Also, in the rest of this chapter I will shorten **true** and **false** by **t** and **f**, respectively.

The semantics of constraints are defined in terms of *valuations*, which are maps from entities to truth values.

Definition 97

1. A *valuation* is map $\nu \in \mathit{Entity} \Rightarrow \mathbb{B}$. I will denote the set of valuations by *Valuation*. □

2. For a valuation ν and constraint c with $\mathscr{D}\mathrm{c} \subseteq \mathscr{D}\nu$, I define the semantics of c w.r.t. ν, denoted $[\![\mathrm{c}]\!]_\nu$, inductively by

$$\begin{array}{rcl} [\![\mathrm{e}]\!]_\nu & \overset{\text{def}}{=} & \nu e \\ [\![\mathrm{c} \Rightarrow \mathrm{c}']\!]_\nu & \overset{\text{def}}{=} & \begin{cases} \mathbf{t}, & \text{if } [\![\mathrm{c}]\!]_\nu = \mathbf{f} \text{ or } [\![\mathrm{c}']\!]_\nu = \mathbf{t} \\ \mathbf{f}, & \text{otherwise.} \end{cases} \\ [\![\neg\mathrm{c}]\!]_\nu & \overset{\text{def}}{=} & \begin{cases} \mathbf{t}, & \text{if } [\![\mathrm{c}]\!]_\nu = \mathbf{f} \\ \mathbf{f}, & \text{otherwise.} \end{cases} \end{array}$$

3. The semantics of constraints are lifted to sets of constraints $\mathrm{C} = \{\, \mathrm{c}^n \,\}$ in the natural way, that is,

$$[\![\mathrm{C}]\!]_\nu \overset{\text{def}}{=} \begin{cases} \mathbf{t}, & \text{if } \forall \mathrm{c} \in \mathrm{C} \ (\ [\![\mathrm{c}]\!]_\nu = \mathbf{t}\ ) \\ \mathbf{f}, & \text{otherwise.} \end{cases}$$

□

A valuation can be obtained from configuration by treating **selected** as $\mathbf{t}$ and **deselected** as $\mathbf{f}$, ignoring entities that are **unspecified**.

Definition 98 If a is a configuration, then its valuation ν_a is

$$\nu_a \overset{\text{def}}{=} \{ e \mapsto \mathbf{t} \mid ae = \textbf{selected} \} \cup \{ e \mapsto \mathbf{f} \mid ae = \textbf{deselected} \} \quad .$$

A valuation ν_a is *total* iff $\mathscr{D}\nu_a = \mathscr{D}a$. □

I have thus obtained a method to accept or reject total configurations, that is, given $C \subseteq$ *Constraint* and total $a \in$ *Configuration*, the configuration a can be accepted iff $[\![C]\!]_{\nu_a} = \mathbf{t}$. But a typical process involving selections and deselections starts with an empty configuration and the goal of the purchaser is to progress towards a total configuration so that the purchase can be made. I therefore need to define a concept of progress.

Definition 99

1. For two specifications s and s', I will say that s *is strictly refined by* s', denoted $s < s'$, iff $s =$ **unspecified** and $s \neq$ **unspecified**. As usual, $\leq$ is the reflexive extension of $<$.
2. For two configurations a and a', I will say that a *is refined by* a', denoted $a \leq a'$, iff $\mathscr{D}a = \mathscr{D}a' \wedge \forall e \in \mathscr{D}a \; (\, ae \leq a'e \,)$.
3. If $a \leq a'$ and $a \neq a'$, then $a < a'$. □

The purchaser thus makes progress towards obtaining a total configuration if he moves from a configuration a to a configuration a' such that $a < a'$. However, real progress is not made unless the configurations are *consistent* with the constraints: A configuration is consistent if it respects all the constraints.

Definition 100 A configuration a is *consistent* w.r.t. a contraint set $C \subseteq$ *Constraint* iff there exists a total configuration a' such that $a \leq a'$ and $[\![c]\!]_{\nu_{a'}} = \mathbf{t}$. □

The final order from a purchaser to an online-shop is always a total and consistent configuration. In the following I will call the system that the purchaser interacts with for the *configurator*, and the setup is thus as follows:

1. The configurator confronts the purchaser with entities.

2. The purchaser may select or deselect entities.

3. When the purchaser selects or deselects an entity, the constraints on the set of entities are checked by the configurator to ensure real progress.

4. If a total and consistent configuration is obtained, the purchaser can submit the order.

12.2 Inference from constraints

When a purchaser selects or deselects an entity e, thus obtaining a configuration a, there are two mutually exclusive situations: Either a is consistent or a is inconsistent (w.r.t. the constraints).

In the latter case, there is no way for the purchaser to select or deselect entities which have so far been left unspecified to progress towards a total, consistent configuration. The response from the configurator could therefore range from '**fail**' to an elaborate explanation of why that particular setting could not be made. For the moment, I will allow the configurator simply to return '**fail**'.

In the former case (a is consistent), there must exist a non-empty set $S = \{ a^n \}$ of total configurations, all satisfying the constraints. If S consists of a single configuration, the purchaser will definitely like to know what that configuration is, so as to complete the purchase. Even if S comprises several configurations, the purchaser will like to know if the setting of any entities are *forced*, in the sense that their settings are a consequence of the selections made so far. But how do I infer such configurations that incorporate forced settings, and is there a *best* such configuration? Indeed there is, since *Configuration* and $\leq$ form a semi-lattice.

Definition 101 A *semi-lattice* $\langle S , \leq \rangle$ is set S ordered by a reflexive, anti-symmetric and transitive ordering $\leq$ such that, for any pair of elements, there is a *greatest lower bound* t, that is,

$$\forall s, s' \in S \; (\exists t \in S \; (s \geq t \leq s' \wedge \forall t' \in S \; (s \geq t' \leq s' \Rightarrow t' \leq t))) \; .$$

The greatest lower bound is denoted $s \sqcap s'$ and the greatest lower bound of a non-empty set $S' \subseteq S'$ is denoted $\bigsqcap S'$. $\square$

Lemma 43 *$\langle$Configuration, $\leq\rangle$ is a semi-lattice where*

$$a \sqcap a' = a \cap a' \cup \{\, e \mapsto \textbf{\textit{unspecified}} \mid ae \neq a'e \,\}$$

and the bottom element of the lattice is the empty configuration.

PROOF. It is easily verified that $\leq$ is reflexive, anti-symmetric and transitive, and that the empty configuration is a lower bound for all configurations. To show that $a \sqcap a' = a \cap a' \cup \{\, e \mapsto \textbf{unspecified} \mid ae \neq a'e \,\}$ is a greatest lower bound, assume that there is a greater lower bound a'' such that $(a \sqcap a') < a''$, which would imply that there existed an $e \in \mathit{Entity}$ such that $(a \sqcap a')e = \textbf{unspecified}$ and $a''e \neq \textbf{unspecified}$. But then either $ae = ae = \textbf{unspecified}$ or $ae \neq ae$, in which case $a'' \not\leq a$ or $a'' \not\leq a'$, and thus not a lower bound, a contradiction. $\square$

So if configration a is consistent w.r.t. constraint set C, the configuration

$$a' = \bigsqcap \{\, a'' \mid a \leq a'' \wedge a'' \text{ is total} \,\}$$

is optimal, in the sense that $a \leq a'$ and there are no C-consistent configurations a'' sucht that $a' < a''$.

To sum it up, when given a configuration a, a configurator should conceptually calculate the set S of all total and consistent refinements of a, and either return an error if $S = \varnothing$, or infer the greatest lower bound $\bigsqcap S$ if $S \neq \varnothing$.

Definition 102 A *configuration-inference algorithm* is a function

$$I \in \mathscr{P}(\mathit{Constraint}) \Rightarrow \mathit{Configuration} \Rightarrow (\mathit{Configuration} + \textbf{fail})$$

such that

$$I\ C\ a = \begin{cases} \bigsqcap S, & \text{if } \varnothing \neq S = \{\, a' \mid a \leq a' \wedge a' \text{ total and C-consistent} \,\} \\ \textbf{fail}, & \text{otherwise.} \end{cases}$$

$\square$

Remark 16 The configuration-inference problem is as hard as the general *satisfiability problem* (SAT), so a configuration-inference algorithm may not be feasible.

It only remains to show how such an algorithm can be derived. In the next sections I will present two solutions to the interactive constraint-solving problem. The first is an indirect approach where program-transformation techniques are used to carry out the the inference. The second is a direct approach where *binary-decision diagrams* are used to perform the inference. Lastly, I will discuss how a configuration system might be designed around the direct approach.

12.3 A program-transformation approach

In the previous chapter I presented a program inversion technique called *explicitation.* This technique can invert a first-order functional program with respect to a particular output. In general, the result of an inversion is a grammar that approximates the set of inputs that would result in the particular output. My first attempt to obtain an inference algorithm is by inverting a program that checks whether a constraint set is true in a given valuation:

```
data Term = And Term Term | Or Term Term | Not Term
          | Imply Term Term | Val Bool
eval (And x y) = if eval x then eval y else false
eval (Or x y) = if eval x then true else eval y
eval (Imply x y) = if eval x then eval y else true
eval (Not x) = if eval x then false else true
eval (Val x) = x
```

(12.1)

This program expects as input a constraint set C in which the entities have been replaced with boolean values according to a total valuation ν. It returns `true` or `false` depending on the value of $\llbracket C \rrbracket_\nu$.

Now, if I invert the checker program w.r.t. the output `true` and a constraint set in which the entities have been replaced by program variables, the result will be a description of all total ν such that $\llbracket C \rrbracket_\nu = \mathbf{t}$.

Example 50 Consider the constraint set

$$\begin{array}{rcl} z & \Rightarrow & v \land (y \lor x \lor x_2) \\ x_2 & \Rightarrow & x_1 \land \neg w \\ w & \Rightarrow & v \lor y \lor x \end{array} \tag{12.2}$$

where x, y, z, v, w, x_1 and x_2 are entities. The encoding of the above constraint would be

```
And (Imply (Val z) (And (Val v) (Or (Val y) (Or (Val x) (Val x2)))))
    (And (Imply (Val x2) (And (Val x1) (Not (Val w))))              (12.3)
         (Imply (Val w) (Or (Val v) (Or (Val y) (Val x)))))
```

where `x`, `y`, `z`, `v`, `w`, `x1` and `x2` are program variables. The result of inverting the checker program (12.1) w.r.t. constraint (12.3) will give me a grammar similar to

$$S ::= \langle \mathbf{f},\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t}\rangle \mid \langle \mathbf{f},\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{f},\mathbf{f}\rangle \mid \ldots \mid \langle \mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f}\rangle \ . \tag{12.4}$$

The grammar explicitly describes all consistent valuations: The entities are implicitly represented by a position in the sequence. □

Remark 17 Inversion of the checker program w.r.t. a constraint set in which the entities have been replaced by program variables will always be a grammar that precisely enumerates the possible of the entities. Since the search space is finite, the same result could be obtained by using a logic programming language.

Having obtained the set of consistent valuations, it is a trivial task to translate it into a list of configurations by making the entities explicit and replacing **t/f** by **selected/deselected**. I will call this list of total configurations for *valids*.

Example 51 The grammar in Equation 12.4 would be translated into the list

$$\begin{bmatrix} \{x \mapsto \mathbf{f}, y \mapsto \mathbf{f}, z \mapsto \mathbf{t}, v \mapsto \mathbf{t}, w \mapsto \mathbf{f}, x1 \mapsto \mathbf{t}, x2 \mapsto \mathbf{t}\} \\ \{x \mapsto \mathbf{f}, y \mapsto \mathbf{t}, z \mapsto \mathbf{t}, v \mapsto \mathbf{t}, w \mapsto \mathbf{f}, x1 \mapsto \mathbf{f}, x2 \mapsto \mathbf{f}\} \\ \vdots \\ \{x \mapsto \mathbf{t}, y \mapsto \mathbf{t}, z \mapsto \mathbf{f}, v \mapsto \mathbf{t}, w \mapsto \mathbf{t}, x1 \mapsto \mathbf{t}, x2 \mapsto \mathbf{f}\} \end{bmatrix} .$$

Having obtained the set of valid configurations, the inference algorithm is quite simple.

Algorithm 1

```
$\mathrm{I}_1 \in \mathit{Configuration}\ \mathrm{List} \Rightarrow \mathit{Configuration} \Rightarrow (\mathit{Configuration} + \mathbf{fail})$
$I_1\ valids\ conf =$
    $\mathbf{case}\ \mathit{filter}\ (conf\ \leq)\ valids\ \mathbf{of}\ (c{:}cs) \rightarrow \mathit{foldl}\ (\sqcap)\ c\ cs$
                                        $|\quad []\quad \rightarrow \mathbf{fail}$
```

The functions '*filter*' and '*foldl*' are the standard functions on lists:

$\mathit{filter} \in (\alpha \Rightarrow \mathrm{Bool}) \Rightarrow \alpha\ \mathrm{List} \Rightarrow \alpha\ \mathrm{List}$
$\mathit{filter}\ f\ [] = []$
$\mathit{filter}\ f\ (x : xs) = \mathbf{if}\ f\ x\ \mathbf{then}\ x : (\mathit{filter}\ f\ xs)\ \mathbf{else}\ \mathit{filter}\ f\ xs$
$\mathit{foldl} \in (\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \beta \Rightarrow \alpha\ \mathrm{List} \Rightarrow \beta$
$\mathit{foldl}\ f\ z\ [] = z$
$\mathit{foldl}\ f\ z\ (x : xs) = \mathit{foldl}\ f\ (f\ x\ z)\ xs$

The *valids* list is in general exponential in number of entities in the constraint set, so the space bound on the algortihm is

$$\mathrm{O}(|conf|\,|valids|) = \mathrm{O}(2^{entities}) \ , \tag{12.5}$$

which makes this solution intractable, even though *valids* can be represented compactly as a vector of bit vectors.

Instead of obtaining all concistent valuations explicitly by inverting the checker program, I could *specialise* it with respect to the constraint set to obtain a program which can accept or reject total configurations. If the specialisation is done by the supercompiler described in Chapter 10, the result will be the simple program

```
cond a b c = if c then cond0 a b else true
cond0 d e = if e then if d then false else true else false
main x y z v w x1 x2 =
    if z then if v
         then if y then cond w x1 x2
              else if x then cond w x1 x2
                   else if x2 then cond0 w x1
                        else false else false
    else if x2 then cond0 w x1
         else if w
              then if v then true
                   else if y then true else x
              else true
```

The interpretative overhead from the checker has been completely eliminated, and the program has become a recursion-free, read-once boolean program. In fact, the program can be represented as the *free binary-decision diagram* [123] depicted in Figure 12.1. Every interior node is labelled with a variable and has two “legs”, a solid and a dashed. Selecting a solid leg means assigning the value **t** to the variable, whereas selecting a dashed legs means assigning **f** to the variable. Each path in a the diagram thus represents an assignment of truth values to the variables (i.e., a valuation). If a path leads to the leaf **0**, it means that that particular valuation is inconsistent; conversely, **1** means that the valuation is consistent. If a variable is not mentioned on a particular path, it means that the value can be either **f** or **t**.

The merit of representing the constraint set as a binary-decision diagram is that several total configurations can share common subconfigurations. This is the key observation that pompts me to abandon the program-tranformation-based solution, and instead cast the problem of configuration inference in a binary-decision-diagram setting.

12.4 Inference by BDDs

I will now encode the constraint set as a *reduced ordered binary-decision diagram* (Bryant [12]), from now on simply called BDDs. The advantages of using BDDs are that there exist numerous efficient implementations, and that the inference problem can be solved by using only basic BDD operations, as I will explain below.

It is well known how to construct a BDD from first-order propositional formulæ, although the chosen variable order (i.e., entity order) is crucial for the size of the BDD — in extreme cases a difference between linear and exponential size. But also optimal variable ordering is a hard problem: The best known algorithm runs in $O(n3^n)$, and even improving the variable ordering is NPTIME-complete (see [10, 30]); so heuristics are used in practice. In the following, I will simply assume that a near-optimal variable order has been obtained by some standard method (see, e.g., [68, 88, 27]). Given a constraint set C, I will denote the resulting BDD by bdd_C.

Remark 18 Calculation of a good variable ordering can be done once and for all when the constraint set C is fixed.

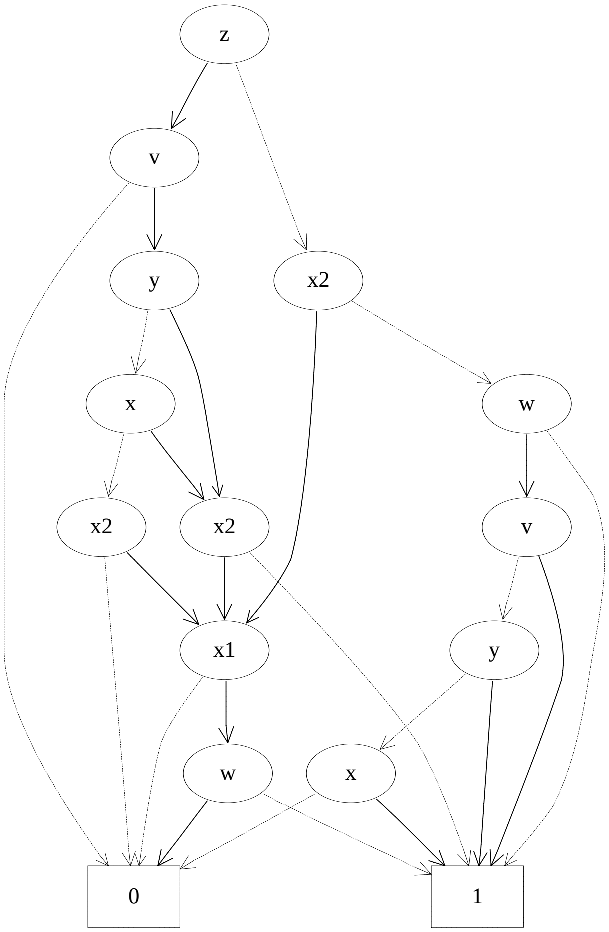


Figure 12.1: Free BDD representation of the specialised constraint validator. Interior nodes are labelled with variables. Each path in a the diagram thus represents an assignment of truth values to the variables: Selecting a solid edge means assigning the value **t** to the variable, and conversely a dashed leg means assigning **f** to the variable. If a path leads to the leaf **0**, it means that that particular assignment results in the value **f**; conversely, **1** means **t**.

Table 12.1: Operations on BDDs.

data *BDD* = $\mathbf{0} \mid \mathbf{1} \mid \ldots$

restrict ∈ *BDD* ⇨ *Valuation* ⇨ *BDD*	O(n)
whentrue ∈ *BDD* ⇨ *BDD*	O(1)
whenfalse ∈ *BDD* ⇨ *BDD*	O(1)
label ∈ *BDD* ⇨ *Entity*	O(1)
$\mathscr{V}$ ∈ *BDD* ⇨ $\mathscr{P}$(*Entity*)	O(n)
exist ∈ *BDD* ⇨ $\mathscr{P}$(*Entity*) ⇨ *BDD*	?

The operations I will need on BDDs are shown in Table 12.1. The *restrict* operation specialises a BDD to a variable assignment, the *whentrue* and *whenfalse* operations select the **t** or **f** branch (if any), respectively, the *label* operations returns the top label of the BDD (if any), and the $\mathscr{V}$ operation that returns the list of variables in a BDD. I refer to the unique unsatisfiable BDD by **0**, and similarly I refer to the unique valid BDD by **1**. Comparing a BDD to **0** or **1** is a constant time operation. An optimal inference algorithm can now be cast as follows.

First specialise a given *bdd* to the given configuration, possibly aborting if the resulting *bdd'* is unsatisfiable. Then deduce impossible variable assignments by speculatively assuming that each variable *e* is **t** or **f**: If the result of, say, *e* = **t** is an unsatisfiable BDD, then *e must* be given the indication **deselected** to fulfil the constraints, that is, the indication of variable *e* is forced. The algorithm below implements this kind of constraint propagation, returning all forced variable indications incorporated into the original configuration.

Algorithm 2

```
$I_2 \in \mathit{BDD} \Rightarrow \mathit{Configuration} \Rightarrow (\mathit{Configuration} + \mathbf{fail})$
$I_2\ \mathit{bdd}\ \mathit{conf} = \mathbf{case}\ \mathit{restrict}\ \mathit{bdd}\ \nu_{\mathit{conf}}$
                 $\mathbf{of}\ \mathbf{0} \to \mathbf{fail}$
                 $\mid \mathit{bdd}' \to \mathit{foldl}\ (\mathit{speculate}\ \mathit{bdd}')\ \mathit{conf}\ (\mathscr{V}\,\mathit{bdd}')$
$\mathit{speculate} \in \mathit{BDD} \Rightarrow \mathit{Entity} \Rightarrow \mathit{Configuration} \Rightarrow \mathit{Configuration}$
$\mathit{speculate}\ \mathit{bdd}'\ e\ \mathit{conf} = \mathbf{if}\ \mathit{restrict}\ \mathit{bdd}'\,\{\, e \mapsto \mathbf{t} \,\} = \mathbf{0}$
                            $\mathbf{then}\ \mathit{conf}_{/e} \uplus \{\, e \mapsto \mathbf{deselected} \}$
                            $\mathbf{else\ if}\ \mathit{restrict}\ \mathit{bdd}'\,\{\, e \mapsto \mathbf{f} \} = \mathbf{0}$
                                $\mathbf{then}\ \mathit{conf}_{/e} \uplus \{\, e \mapsto \mathbf{selected} \}$
                                $\mathbf{else}\ \mathit{conf}$
```

where ν_{conf} is the translation of *conf* into a partial valuation, omitting the entities that are **unspecified**.

Proposition 6 *Algorithm 2 performs configuration-inference.*

PROOF. Let $C \subseteq \mathit{Constraint}$, $a \in \mathit{Configuration}$ and $\mathit{bdd} \in \mathit{BDD}$ be the encoding of C. I need to show that, if the set

$$S = \{\, a' \mid a \leq a' \wedge a' \text{ total and C-consistent} \}$$

is non-empty, then $I_2\ \mathit{bdd}\ a = \bigsqcap S$, and otherwise $I_2\ \mathit{bdd}\ a = \mathbf{fail}$.

If I assume $S = \varnothing$, then all refinements of a are C-inconsistent, and thus any path in $\mathit{bdd}' = \mathit{restrict}\ \mathit{bdd}\ \nu_a$ will lead to $\mathbf{0}$. By definition (Bryant [12]), there is only one BDD with this property, namely $\mathbf{0}$. Hence $\mathit{bdd}' = \mathbf{0}$, and $I_2\ \mathit{bdd}\ a = \mathbf{fail}$ as required.

Otherwise, assume $S = \{\, a^n \,\} \neq \varnothing$ and $\mathit{bdd}' = \mathit{restrict}\ \mathit{bdd}\ \nu_a \neq \mathbf{0}$. To show that $I_2\ \mathit{bdd}\ a = \bigsqcap S$, it is sufficient to show, if $a' = I_2\ \mathit{bdd}\ a$, then, for all e, $a'e$ is the greatest lower bound for e. I now divide the entities $e \in a$: Either $e \in \mathscr{V}\,\mathit{bdd}'$ or $e \notin \mathscr{V}\,\mathit{bdd}'$.

If $e \notin \mathscr{V}\,\mathit{bdd}'$, then either $ae \neq \mathbf{unspecified}$ or $ae = \mathbf{unspecified}$. If $ae \neq \mathbf{unspecified}$, then $\forall a' \in S\ (\, ae = a'e \,)$ since $\forall a' \in S\ (\, a \leq a' \,)$. Otherwise $ae = \mathbf{unspecified}$, and thus the value of e does not affect any path in bdd', which implies that

$$\forall a' \in S \left(\exists a'' \in S \left(\begin{array}{c} a'e = \mathbf{selected} \Rightarrow a''e = \mathbf{deselected} \\ \wedge \\ a'e = \mathbf{deselected} \Rightarrow a''e = \mathbf{selected} \end{array} \right) \right) .$$

In summary, if $e \notin \mathscr{V}\,\mathit{bdd}'$, then ae is the greatest lower bound for e. It therefore remains to show that '*foldl* (*speculate* bdd') a ($\mathscr{V}\,\mathit{bdd}'$)' evaluates to a a' such that $a'e = ae$ for all $e \notin \mathscr{V}\,\mathit{bdd}'$ and $a'e$ is the greatest lower bound for all $e \in \mathscr{V}\,\mathit{bdd}'$. From the definition of *speculate*, it is easy the see that only entites $e \in \mathscr{V}\,\mathit{bdd}'$ are affected by the fold, and furthermore that each such e is affected only once and in isolation. It therefore remains to show that *speculate* bdd' e a gives the greatest lower bound for e.

So, if $e \in \mathscr{V}\,\mathit{bdd}'$, then $ae =$ **unspecified** (because $\mathit{bdd}' = \mathit{restrict}\ \mathit{bdd}\ \nu_a$) and $a'e \neq$ **unspecified** for all $a' \in S$ (because all a' are total). Now, if $\exists b \in \mathbb{B}\ (\forall a' \in S\ (\nu_{a'} e = b\,))$, then *restrict* bdd' $\{e \mapsto \neg b\} = \mathbf{0}$, and thus '*speculate* bdd' e a' will change the corresponding indication for e from **unspecified** to **selected** (if $b = \mathbf{t}$) or **deselected** (if $b = \mathbf{f}$). Otherwise, $\exists\{a', a''\} \subseteq S\ (a'e \neq a''e)$, and e is left unspecified. In summary, *speculate* bdd' e a gives the greatest lower bound for e. □

The running time of the algorithm is

$$O(|\mathit{bdd}| + |\mathit{var}\ \mathit{bdd}|\,|\mathit{bdd}|) \geq O(|\mathit{bdd}| \log |\mathit{bdd}|)\ , \qquad (12.6)$$

so there is certainly need for improvements, considering that the BDDs can be exponential in size of the constraint set. In practice, however, I expect many of the constraints to be unrelated such that a good variable ordering can be found to reduce the size of the corresponding BDDs. For instance, when configuring a computer, the kind of mother board influences the type of processor that can be chosen, but it does not influence whether a CD-ROM, CD-RW or DVD unit can be selected.

12.4.1 Breadth-first traversal

The inference carried out by folding '*speculate*' over the variables of bdd' can be done more efficiently by a single traversal of bdd'.[2] The key observation is that, if

$$\mathit{restrict}\ \mathit{bdd}'\ \{e \mapsto \mathbf{t}\} = \mathbf{0}\ ,$$

then it holds for bdd' that

1. every branch that leads to **1** will contain a node labelled e, and

[2] The function '*speculate*' was suggested by Ken Friis Larsen (IT-C, Denmark) as a conceptual simplification of the more efficient function that follows.

2. every node labelled e must have the $\mathbf{t}$-leg connected to $\mathbf{0}$;

and conversely if *restrict bdd'* $\{e \mapsto \mathbf{f}\} = \mathbf{0}$.

Using the above observations, I can devise an algorithm that calculates the forced variable indications by traversing the BDD breadth-first with the help of priority queue.

Algorithm 3 Let Q be a priority queue that can hold nodes of the BDD such that the order of the nodes are dictated by the order of the variables.[3] Initially, Q contains the single root node.

1. If Q is empty, terminate.
2. Let the label of the first node in Q be x, and let the set $N = \varnothing$. While the label of the first node n in Q is x, remove n from Q and insert n into N.
3. If Q is empty, condition 1 is true. If furthermore all nodes in N have the same leg in $\mathbf{0}$, condition 2 is true, and thus I have found a forced indication for a particular variable; output this indication.
4. Let C be all the children of nodes in N. Remove all $\mathbf{0}$-nodes from C.
5. If C contains $\mathbf{1}$-nodes, the algorithm terminates, since condition 1 cannot be fulfilled by any of the remaining variables.
6. Add C to Q, and repeat step 1. □

In algorithm 3, each node is inserted at most once into Q, and each such insertion takes time $O(\log|Q_{\max}|)$ where $Q_{\max}$ is the largest Q. The running time of algorithm 3 is thus $O(|bdd|\log|Q_{\max}|)$. If I replace the traversal of all the variables in algorithm 2 with the implementaion of algorithm 3 shown in Figure 12.2, the total running time of the resulting algorithm is

$$O(|bdd| + |bdd|\log|Q_{\max}|) \leq O(|bdd|\log|bdd|) \quad , \tag{12.7}$$

which is an improvement on the previous upper bound (12.6).

[3] Recall that the variables in a BDD are totally ordered; the largest variable is the one labelling the root.

```
data Queue a = EMPTY | ...
enqueue ∈ a ⇨ Queue a ⇨ Queue a
dequeue ∈ Queue a ⇨ Queue a
front ∈ Queue a ⇨ a

infer ∈ BDD ⇨ Configuration
infer bdd = if bdd = 0 ∨ bdd = 1 then [] else inf (enqueue bdd EMPTY) ∅
inf EMPTY forced = forced
inf q forced = let ⟨same , q'⟩ = collect (label (front q)) q []
                   whentrues = map whentrue same
                   whenfalses = map whenfalse same
                   children = append whentrues whenfalses
                   forced' = if q' ≠ EMPTY then forced
                             else if all (= 0) whentrues
                                  then {x ↦ f} ∪ forced
                                  else if all (= 0) whenfalses
                                       then {x ↦ t} ∪ forced
                                       else forced
               in inf (inject children q) forced'
collect x q same = if q ≠ EMPTY ∧ x = label (front q)
                   then collect x (dequeue q) (front q : same)
                   else ⟨same , q⟩
inject [] q = q
inject (bdd : bdds) q = if bdd = 1 then EMPTY
                        else if bdd = 0 then inject bdds q
                             else inject bdds (enqueue bdd q)
all p  = foldl ((∧) ∘ p) t
```

Figure 12.2: Inference on BDDs.

12.4.2 Partitioning the constraint set

The upper bound $O(|bdd_C| \log |bdd_C|)$ is still not tractable, since $|bdd_C| \leq 2^{|C|}$. I might therefore look for properties of the constraint set that can make the inference more tractable. In particular, I could aim at building a set of smaller BDDs from the constraint set, instead of building a single large BDD. That is, assuming $C \subseteq \mathit{Constraint}$ and $a \in \mathit{Configuration}$, I am looking for a partitioning $\{ C^n \}$ of C and an inference algorithm I_4 such that

$$\begin{array}{rcl} C & = & C_1 \wedge \cdots \wedge C_n \\ & \wedge & \\ a & \preceq & I_4 \ \langle bdd_{C_1}, \ldots, bdd_{C_n} \rangle \ a \quad \preceq \quad I_2 \ bdd_C \ a \end{array} \qquad (12.8)$$

where

$$a \overset{\text{def}}{\preceq} a' \quad \text{iff} \quad a \leq a' \vee a' = \mathbf{fail} \ .$$

The use of $\preceq$ in Equation 12.8 allows me to select a partitioning of the constraint set which leads to non-optimal inference.

Remark 19 *Any* partitioning of the constraint set will fulfil Equation 12.8, so choosing a particular partitioning only affects the precision of the inference.

Clearly, if the constraint set can be divided into *independent* parts, in the sense that each pair of parts have no variables in common, Equation 12.8 will hold even if the last occurrence of $\preceq$ is replaced by $=$, and thus optimality is regained. Partitioning into independent parts has apparently been used successfully in the hardware-varification community. At the present time, I have not been able to find better partioning schemes which ensure optimality.

Algorithm 4 shows a simple implementation of I_4 in which Algorithm 2 is repeatedly applied to each part C_i, accumulating new configurations until a fixpoint is reached.

Algorithm 4

```
$I_4 \in \mathit{BDD}\ \mathrm{List} \Rightarrow \mathit{Configuration} \Rightarrow (\mathit{Configuration} + \mathbf{fail})$
$I_4\ \mathit{bdds}\ a = \mathit{fix}\ (\mathit{runthrough}\ \mathit{bdds})\ a$
where $\mathit{fix}\ f\ d = \mathbf{let}\ d' = f\ d\ \mathbf{in}\ \mathbf{if}\ d = d'\ \mathbf{then}\ d\ \mathbf{else}\ \mathit{fix}\ f\ d'$
      $\mathit{runthrough}\ \mathit{bdds}\ a = \mathit{foldl}\ \mathit{propagate}\ a\ \mathit{bdds}$
      where $\mathit{propagate}\ \mathbf{fail}\ \ \mathit{bdd} = \mathbf{fail}$
            $\mathit{propagate}\ a\ \mathit{bdd} = I_2\ \mathit{bdd}\ a$
```

12.4.3 Incremental BDDs

There is no need to re-specialisation of the BDD representation of the constraint set with every user interaction: If a configuration a' is a refinement of the previously configuration a, then the BDD *bdd* corresponding to the specialisation of the constraint set to a can be re-used when checking the configuration a'.

Lemma 44 *Let* $C \subseteq$ *Constraint. Then, for all* $\{\, a\,,\, a' \,\} \subseteq$ *Configuration,*

$$a \leq a' \Rightarrow \mathit{restrict}\ \mathit{bdd}_C\ \nu_{a'} = \mathit{restrict}\ (\mathit{restrict}\ \mathit{bdd}_C\ \nu_a)\ \nu_{(a' \setminus a)} \ .$$

PROOF. Assume $a \leq a'$. Then

$$\forall e \in \mathscr{D}a\ (\, ae = a'e \vee (ae = \textbf{unspecified} \wedge a'e \neq \textbf{unspecified})\,) \ ,$$

and thus $\nu_a \subseteq \nu_{a'}$ which implies $\nu_a \between \nu_{(a' \setminus a)}$. By definition (Bryant [12]),

$$\nu \between \nu' \Rightarrow \mathit{restrict}\ \mathit{bdd}\ (\nu \cup \nu') = \mathit{restrict}\ (\mathit{restrict}\ \mathit{bdd}\ \nu)\ \nu' \ ,$$

and hence

$$\begin{aligned} \mathit{restrict}\ \mathit{bdd}_C\ \nu_{a'} &= \mathit{restrict}\ \mathit{bdd}_C\ (\nu_a \cup \nu_{(a' \setminus a)}) \\ &= \mathit{restrict}\ (\mathit{restrict}\ \mathit{bdd}_C\ \nu_a)\ \nu_{(a' \setminus a)} \ . \end{aligned}$$

□

I can therefore avoid unnecessary re-specialisation of the initial constraint set B_C in Algorithm 2 (and thus Algorithm 4) by remembering the last configuration and corresponding BDD between user interactions. The constraint set thus decreases in size with each refinement. Of course, when the inference engine receives a configuration that is *not* a refinement of the previous one, it is necessary to restart and use the initial representation bdd_C of C again. With such a setup, only the difference between configurations needs to be transmitted between the purchaser and the inference engine. This setup is depicted in Figure 12.3.

Another merit of using the BDDs incrementally is that default indications for all entities could be represented as a total and consistent configuration a_{default}: When the user has finished his selections, in the sense that he has obtained a consistent but not necessarily total configuration a, the defaults a_{default} are used to further specialise the constraint set. Since no user-(de)selected entity is mentioned in the BDD corresponding to a, the default indication for such an entity is effectively ignored; only entities deferred by the user will be affected by the default indications.

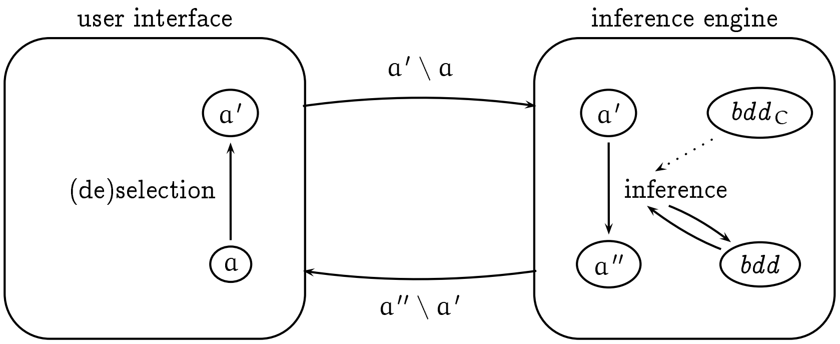


Figure 12.3: Incremental configuration. The difference between the old and new configuration is transmittet to the inference engine, which then reconstructs the new configuration and propagates forced settings inferred from either the previous or the initial BDD, depending on whether the the new configuration is a refinement of the old or not.

12.4.4 Explanations

Providing the user with an explanation of why a particular selection is unsatisfiable is also easier when the algorithm maintains a state between user interactions, as explained in the previous section.

Consider the situation where the configuration a was accepted, but the following a' is not accepted. The fact that a was accepted can be used to narrow the explanation to the set of settings that upset the configurator, because of the following property.

Lemma 45 *If* $\{ a , a' \} \subseteq$ *Configuration and bdd* $\in$ *BDD, then*

$$\textit{restrict bdd } \nu_a \neq \boldsymbol{0} \Rightarrow \textit{restrict bdd } \nu_{a \sqcap a'} \neq \boldsymbol{0} \ .$$

PROOF. Assume that *restrict bdd* $\nu_a \neq \mathbf{0}$ but *restrict bdd* $\nu_{a \sqcap a'} = \mathbf{0}$. Then there exists at least one total configuration a'' such that $a \leq a''$ and *restrict bdd*

$v_{a''} = \mathbf{1}$. Since $(a \sqcap a') \leq a$, also $(a \sqcap a') \leq a''$. But then, by Lemma 12.4.3,

$$\begin{aligned}\mathbf{1} &= \mathit{restrict}\ \mathit{bdd}\ v_{a''} \\ &= \mathit{restrict}\ (\mathit{restrict}\ \mathit{bdd}\ v_{a \sqcap a'})\ v_{a'' \setminus (a \sqcap a')} \\ &= \mathit{restrict}\ \mathbf{0}\ v_{a'' \setminus (a \sqcap a')} \\ &= \mathbf{0}\ ,\end{aligned}$$

a contradiction. □

If I thus specialise the initial constraint set C to the greatest lower bound of the two configurations $\mathit{bdd}_{\sqcap} = \mathit{restrict}\ \mathit{bdd}_C\ v_{a \sqcap a'}$, the variables that do not contribute to the failure of configuration a', namely $\{ e \mid a'e = \mathbf{unspecified} \}$, can be removed from $\mathit{bdd}_{\sqcap}$ by *existential qualification*:

$$\mathit{bdd}_{\mathrm{explanation}} = \mathit{exist}\ \mathit{bdd}_{\sqcap}\ \{ e \mid a'e = \mathbf{unspecified} \}\ . \tag{12.9}$$

The '*exist*' operations leaves only the troublesome variables, and thus $\mathit{bdd}_{\mathrm{explanation}}$ is a minimal explanation of why configuration a' could not be accepted.

Example 52 Consider the constraint set in Equation 12.2 and assume the user has first made the selections

$$\{ x \mapsto \mathbf{deselected}\, ,\ y \mapsto \mathbf{selected}\, ,\ (\text{...the rest } \mathbf{unspecified}) \}$$

which succeeds. Assume now that the user changes his mind w.r.t. y and makes additional selection

$$\{ x \mapsto \mathbf{deselected}\, ,\ y \mapsto \mathbf{deselected}\, ,\ z \mapsto \mathbf{selected}\, ,\ x_2 \mapsto \mathbf{deselected}\, ,\ (\dots) \}$$

which leads to failure. The msg of the two configurations is simply

$$\{ x \mapsto \mathbf{deselected}\, ,\ (\text{...the rest } \mathbf{unspecified}) \}$$

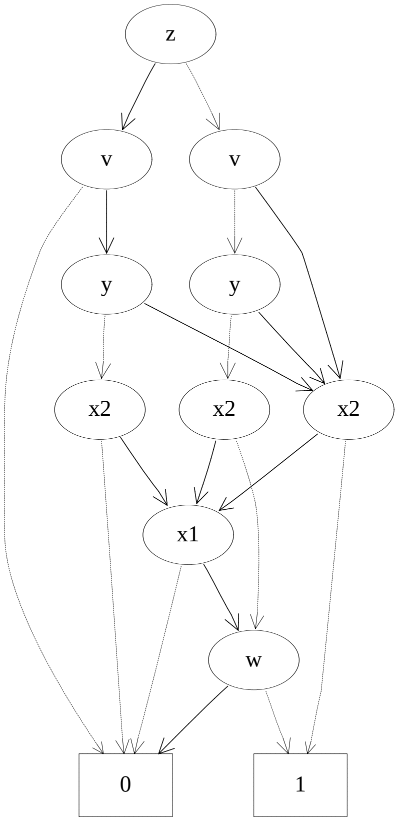
and the set of unspecified variables is $\{ v\, , w\, , x_1 \}$. The resulting $\mathit{bdd}_{\sqcap}$ and $\mathit{bdd}_{\mathrm{explanation}}$ can be seen in Figure 12.4. $\mathit{bdd}_{\mathrm{explanation}}$ could then easily be converted to

$$z \Rightarrow (\neg y \Rightarrow x2)\ ,$$

which can be post-processed to the more understandable

$$z \Rightarrow y \vee x2\ .$$

$bdd_{\sqcap} =$
$restrict\ bdd_{C}\ \{x \mapsto \textbf{deselected}\} =$



$bdd_{\text{explanation}} =$
$exist\ bdd_{\sqcap}\ \{v\,,\ w\,,\ x_1\} =$

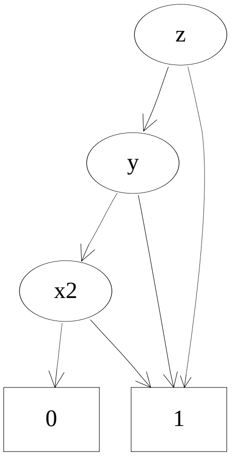


Figure 12.4: Explanation of unsatisfiable constraints. The explanation BDD captures the relationship between variables z, y and x_2 which led to the non-accepting situation.

12.5 Related Work

There exist a significant literature on configuration, in particular a body of papers that view the configuration problem as a *constraint-satisfaction problem*, see e.g., Sabin and Weigel's survey [91] or Veron, Fargier and Aldanondo [119].

A more thorough analysis of why the approach presented here is sufficient for most cases can be found in [107]. In this paper we describe how most real-world concerns pertaining to configuration can be expressed in our simple framework (i.e., how aggregation, inheritance, etc. can be translated into simple constraints).

The company ConfigIt Software (www.configit-software.com) has developed a so-called *virtual table* technique (See Møller, Andersen and Hulgaard [72]) that seems to address the inference problems I have stated here. Their technique is not well described, however, since they have a pending patent on it.

The inference algorithm I_2 on BDDs is akin to the Dilemma Rule used in Stålmarck's proof procedure for propositional logic [100].

Chapter 13

Summary

In Part I of this thesis, I have presented some of the most influential driving-based program-tranformation techniques for first-order functional programs in a unified framework. The framework is build around the notion of a *transformation trace tree* obtained by speculatively executing the source code of a program. The speculative execution is carried out by a kind of symbolic evaluation called *driving* which, in addition to normal computation, speculatively instantiates variables that block computations from proceeding. The instantiations are derived from the source code of the program in such a way that they, as a whole, form a complete description of the values that variables can be bound to. The difference between traditional *deforestation*, *generalised partial computation* and various notions of *supercompilation* lies in what kind of instantiations are performed, and how the instantiations are propaged to the resulting branches in the transformation trace tree.

Deforestation propagates no information.

Supercompilation propagates positive (and possibly negative) information and uses this information to control the speculative execution.

GPC potentially propagates all kinds of information about primitive operators and data structures, as well as instantiations, by means of a theorem prover.

Tupling pre-calculates which function calls that can be tupled together.

This part of the thesis has been centred around acyclic directed graphs (dags). I have presented a framework based on dag rewriting, as opposed to the term rewriting approaches described in Part I. The dag-based framework makes it possible reason about the semantics of programs and fragments thereof, without having to take the actual representation of terms into account. On top of this framework, I have built a program-transformer which, when using *fully-collapsed* dags, performs all of the essential optimisations seen in deforestation, supercompilation and tupling, as well as providing a guarantee of termination of the transformation process itself. The framework also expresses program transformation in a more realistic setting, namely that of graph rewriting, the standard way to implement non-strict functional languages. To show that it is indeed possible to perform the presented transformation technique in practice, I have run a few examples through a prototype implementation. As a more exotic application of program transformation, I have described how dag-based program-transformation can be used to invert programs. The inverted programs are expressed as *dag grammars* that describe the set inputs giving a particular output. Lastly, I have shown how program transformation can be used for interactive product-line configuration, which in turn establishes an interesting relation between program transformation using fully-collapsed dags and binary-decision diagrams.

13.1 Critical assessment

I set out to address problems with existing driving-based program-transformation techniques with the help of acyclic directed graphs. I believe to have succeeded in some respects, namely in formulating and implementing a supercompiler that is guarenteed to terminate on all programs, and which generally makes programs run faster.

However, I have not succeeded in proving correctness of the transformed programs (w.r.t. the semantics of the original), nor identified the class of programs that will not suffer a slowdown. Moreover, despite the pains I have taken to make the online non-termination detection heuristic (homeomorphic embedding) fast, it is almost certain that the described program-transformation technique will not scale, and thus be usable in practice.

13.2 Further Work

Apart from amending the previously described flaws in the theoretical framework, I see the following lines of future work.

13.2.1 Off-line annotation

As previously mentioned, the online non-termination detection based on homeomorphic embedding is inherently inefficient. To make the driving-based methods applicable in real compilers, it is imperative that most decisions about when to generalise be made off-line. The dag grammars presented in Section 11.2 might be a good vehicle for expressing abstract data and control flow properties of programs, using the techniques presented by Sørensen [105].

A variation of this off-line theme is to identify which programs can safely be driven without the need for generalisations. As seen in Section 10.3.4, supercompilation of a small interpreter specialised to a particular program effectivly removes the interpretation overhead without the need for generalisation.

13.2.2 Re-evaluating generalisations

Although non-termination detection is achieved by considering the whole model (transformation trace) of the program during transformation, generalisations are performed locally and never re-evaluated. The problem with this strategy is that, in practice, it often happens that some of the sub-terms resulting from a generalisation unfold to passive terms; when these passive terms are re-inserted into the generalised term, driving of the whole term can be resumed because the term no longer poses a problem with respect to non-termination.

13.2.3 Higher-order programs

The object language treated in this thesis does not accommodate the needs of real-life programmers. Letting higher-order functions into the language would amend this shortcoming by allowing programs to be build in modular way, which would greatly improve applicability of the language. It seems manageable to extend the theory with higher-order functions as follows.

If I allow curried application of named functions, but not λ-expressions, I can treat curried function applications as passive terms, in the sense that their unfolding is delayed until fully applied. Such an approach is possible because curried function applications will never be *needed* redexes because both pattern matching and primitive operators require their arguments to be of ground type. Since λ-expressions can always be removed by λ-lifting, the only remaining issue with higher-order functions is thus what to with terms of the form

$$g\ (x\ t^n)\ u^m\ ,$$

where $g \in$ *Matcher* and $x \in$ *Variable*. A poor man's solution would be to generalise such terms into

$$\textbf{let}\ y := x\ t^n\ \textbf{in}\ g\ y\ u^m$$

and treat x as a constructor, that is, leave it alone and simply transform the children t^n.

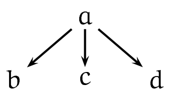
Moreover, if higher-order constructs were allowed in the language, it would be feasible to implement the supercompiler in its own object language. Having done this, it will be possible to experiment with self-application as described by Jones, Sestoft, and Søndergaard [48] or Turchin [117].

Appendix A

Notation

I use the so-called *Currying* notation, which means that when I write "$(a\ b\ c\ d)$" in a formula, it can mean one of two things.

1. If a is a *constructor* (i.e., a symbol with no special interpretation), then "$(a\ b\ c\ d)$" means the *tree* consisting of a root labelled a and ordered children b, c, and d:

2. If a is a symbol interpreted as a function, then "$(a\ b\ c\ d)$" means "apply a to b; apply the result to c; and finally apply the latter result to d.", that is, the *result* (if any) of $((a\ b)\ c)\ d$.

I will use *infix* and *surroundfix* notation like "$(x \in S)$", "$(\langle a\ ,\ b\ ,\ c\ ,\ d\rangle)$" and "$(\Delta \vdash \alpha \rightsquigarrow \beta)$" for convenience, even though it really should be written something like "$(\in x\ S)$", "$(\langle\rangle\ \ a\ b\ c\ d)$", and "$(\vdash_{\rightsquigarrow}\ \Delta\ \alpha\ \beta)$".

In general, parentheses and commas in formulæ will *only* be used as meta-syntactical delimiters to group or separate objects, that is, to avoid ambiguous interpretations. When convenient, I will leave out parentheses and commas.

I will often need to write sequences such as $x_1\ x_2\ x_3\ x_4\ x_5$, and I will therefore introduce the shorthand notation $(x_.)^5$ for such a sequence: The superscript "5" denotes that the preceding syntactic object "$x_.$" should be replicated five times, with the dot replaced by the consecutive numbers 1, 2, 3, 4 and 5. If the replicated object is syntactically simple, I will leave out the dot all together, and, for example, write x^n instead of $(x_.)^n$. When this kind of notation is used in several layers, the innermost part is expanded first. Hence

$$\left\{\, (x_. \mapsto t^2)^n \,\right\} = \{ x_1 \mapsto t_1\ t_2 \,, \ldots \,, x_{n-1} \mapsto t_1\ t_2 \,, x_n \mapsto t_1\ t_2 \} \enspace .$$

The empty sequence is also allowed, so

$$\left\{\, (x_. \mapsto t_.)^0 \,\right\} = \{\, \} \enspace .$$

I will use the symbol $\stackrel{\text{def}}{=}$ when I define things, and I will write "iff" as a shorthand for "if, and only if".

A.1 Sets

I let $S = \{ a \,, b \,, c \,, d \}$ denote that S is a set containing the objects a, b, c, and d (and nothing else). I use the standard infix notation for *membership* ($\in$), *subset* ($\subseteq$), *union* ($\cup$), and *difference* ($\setminus$), where

$$S \setminus S' \stackrel{\text{def}}{=} \{ s \mid s \in S \wedge s \notin S' \} \enspace .$$

When two sets S and T are *disjoint*, I write $S \lightning T$. The *disjoint union* ($\uplus$) is like union, except that it is only defined when the operands are disjoint, that is,

$$S \uplus T \stackrel{\text{def}}{=} \begin{cases} S \cup T, & \text{if } S \lightning T \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

I use the following standard sets.

$$\begin{aligned} \varnothing &\stackrel{\text{def}}{=} \text{empty set} \\ \mathbb{B} &\stackrel{\text{def}}{=} \text{booleans } \{ \text{false} \,, \text{true} \} \\ \mathbb{N}_1 &\stackrel{\text{def}}{=} \text{natural numbers } \{ 1 \,, 2 \,, 3 \ldots \} \\ \mathbb{N} &\stackrel{\text{def}}{=} \text{natural numbers incl. nought } \{ 0 \,, 1 \,, 2 \,, \ldots \} \\ \mathbb{R} &\stackrel{\text{def}}{=} \text{real numbers } \left\{ \ldots \,, -\tfrac{1}{3} \,, \ldots \,, \sqrt{2} \,, \ldots \right\} \end{aligned}$$

Given a set S, $|S| \in \mathbb{N} \cup \{ \infty \}$ denotes the cardinality of S. The set of all subsets of S (the *power set* of S) is denoted $\mathscr{P} S$. I also use shorthand notation for sets. For example, $\{ x_0 \,, x^n \}$ means $\{ x_0 \,, x_1 \,, \ldots \,, x_n \}$.

A.2 Ordered lists

I let $\langle a\,,\, b\,,\, c\,,\, d\rangle$ denote the *ordered list* consisting of the four objects a, b, c, and d, from left to right. If L is a list, I will write $s \in L$ for list membership, and write $|L| \in \mathbb{N} \cup \{\,\infty\,\}$ for the length of L. I write $L \mathbin{+\!\!+} L'$ for the concatenation of L' to the end L, defined as

$$L \mathbin{+\!\!+} L' \overset{\text{def}}{=} \begin{cases} L, & \text{if } |L| = \infty \\ L', & \text{if } |L| = 0 \\ \langle s\rangle \mathbin{+\!\!+} (L'' \mathbin{+\!\!+} L'), & \text{if } L = \langle s\rangle \mathbin{+\!\!+} L'' \end{cases} ,$$

and I write $L \div L'$ for the substraction of L' from L, defined as the result of removing all elements of L' from L, that is,

$$\begin{aligned} \langle\rangle \div L' &\overset{\text{def}}{=} \langle\rangle \\ (\langle s\rangle \mathbin{+\!\!+} L) \div L' &\overset{\text{def}}{=} \begin{cases} L \div L', & \text{if } s \in L' \\ \langle s\rangle \mathbin{+\!\!+} (L \div L'), & \text{otherwise.} \end{cases} \end{aligned}$$

I write $L' \leq L$ iff $L = L' \mathbin{+\!\!+} L''$ for some L'', and I use $L[n]$ to denote the n'th element of list L, counting from 0. The set of finite lists of elements taken from set S is denoted $S^{\star}$, whereas the set of finite and infinite lists of elements from S is denoted S^{ω}.

If a list is finite I will call it a *tuple*. Given two sets S and T, the set of two-tuples over S and T is denoted $S \times T$ (and extended to n-tuples in the obvious way). I also use shorthand notation for tuples. For example, $\langle x_0\,,\, x^n\rangle$ means $\langle x_0\,,\, x_1\,,\, \ldots\,,\, x_n\rangle$.

A.3 Relations

For a relation $\rightarrow \subseteq S \times T$, I write $s \rightarrow t$ as a shorthand for $\langle s\,,\, t\rangle \in \rightarrow$, and I let $s \not\rightarrow t$ mean $\langle s\,,\, t\rangle \notin \rightarrow$. I define the domain $\mathscr{D}\!\rightarrow \overset{\text{def}}{=} \{\, s \mid \exists t\ (\, s \rightarrow t\,)\}$ and the range $\mathscr{R}\!\rightarrow \overset{\text{def}}{=} \{\, t \mid \exists s\ (\, s \rightarrow t\,)\}$. The *restriction* of $\rightarrow$ to a domain S' is denoted $\lceil\rightarrow\rceil_{S'}$ and defined as $\lceil\rightarrow\rceil_{S'} \overset{\text{def}}{=} \{\,\langle s\,,\, t\rangle \mid s \in S' \wedge s \rightarrow t\,\}$. An asymmetric relational symbol has an inverse: $\leftarrow \overset{\text{def}}{=} \rightarrow^{-1}$. I say that $\rightarrow$ is *deterministic* iff

$$\forall s \in S\ (\, s \rightarrow t \wedge s \rightarrow t' \Rightarrow t = t'\,) \quad ,$$

that is, $\rightarrow$ is a (partial) function. To denote that f is a (partial) function, I write $f \in S \mathrel{⇨} T$, and I use prefix notation: $f\ s = t$ means $\langle s\,,\ t\rangle \in f$. If the domain of f is finite, I will use $\{\,(s_. \mapsto t_.)^n\,\}$ to denote the set of bindings that f comprises. I use $\circ$ as infix operator on functions to denote composition: $(f \circ g)\ x \overset{\text{def}}{=} f\ (g\ x)$. If $\rightarrow$ is a binary relation on $S \times S$, I denote by $\xrightarrow{+}$ the transitive closure of $\rightarrow$, and by $\xrightarrow{*}$ the reflexive closure of $\xrightarrow{+}$. The normal forms of $\rightarrow$ is the set $\rightarrow\text{-nf} \overset{\text{def}}{=} S \backslash (\mathscr{D}\!\rightarrow)$. Composition of binary relations is denoted by juxtaposition:

$$\forall \underset{A}{\rightarrow}, \underset{B}{\rightarrow}\ \left(\underset{A}{\rightarrow}\underset{B}{\rightarrow} \overset{\text{def}}{=} \{\, \langle s\,,\ s'\rangle \mid \exists s''\ (\, s \underset{A}{\rightarrow} s'' \wedge s'' \underset{B}{\rightarrow} s'\,)\,\} \right) \ .$$

I write $\xrightarrow{n}$ for the n-fold composition of $\rightarrow$. For all "less-than" relations $\prec\ \subseteq S \times S$, I implicitly define $\preceq$ to be the reflexive extension of $\prec$, that is $\preceq\ \overset{\text{def}}{=}\ \prec \cup \{\, \langle s\,,\ s\rangle \mid s \in S\,\}$.

A.4 Trees

Given a set S, a *tree over* S is a partial map $\tau \in \mathbb{N}_1^{\star} \mathrel{⇨} S$ from sequences of positive natural numbers to S such that

1. τ is non-empty ($\mathscr{D}\tau \neq \varnothing$)

2. τ is prefix closed ($\eta \mathbin{+\!\!+} \mu \in \mathscr{D}\tau \Rightarrow \eta \in \mathscr{D}\tau$)

3. τ is finitely branching ($\eta \in \mathscr{D}\tau \Rightarrow \{\, i \mid \eta \mathbin{+\!\!+} \langle i\rangle \in \mathscr{D}\tau\,\}$ is finite)

4. τ is ordered ($\eta \mathbin{+\!\!+} \langle i\rangle \in \mathscr{D}\tau \Rightarrow \forall j \leq i\ (\,\eta \mathbin{+\!\!+} \langle j\rangle \in \mathscr{D}\tau\,)$)

The domain of a tree is called the *nodes*, and the range is called the *labels*. The node $\langle\rangle$ is called the *root*. For any node $\eta \in \mathscr{D}\tau$, the nodes $\eta \mathbin{+\!\!+} \langle i\rangle$ are called the *children of* η, and η is called the *parent* of these nodes. A tree τ is *singleton* iff $\mathscr{D}\tau = \{\,\langle\rangle\,\}$. A node with no children is a *leaf*, and the set of leaves of τ is denoted leaves_τ. I write $\eta <_\tau \mu$ iff η is a proper ancestor of μ in τ (i.e., $\eta < \mu$ and $\{\,\eta\,,\ \mu\,\} \subseteq \mathscr{D}\tau$). A *branch in* τ is a sequence $\langle \eta_0\,,\ \eta_1\,,\ \ldots\,\rangle$ of nodes from τ such that $\eta_0 = \langle\rangle$ and η_i is the parent of η_{i+1}, for all i. The set of finite trees over S is denoted *Tree* S, whereas the set of finite and infinite trees is denoted $\mathit{Tree}^{\omega}\ S$.

If $\{\tau, \pi\} \subseteq \mathit{Tree}^{\omega}\ S$ and $\eta \in \mathscr{D}\tau$, then $\tau[\eta{:=}\pi]$ denotes the replacement of the subtree rooted at η by the tree π, that is, a tree τ' defined by

$$\begin{aligned} \mathscr{D}\tau' &= (\mathscr{D}\tau \setminus \{\mu \mid \mu \in \mathscr{D}\tau \wedge \eta \leq_{\tau} \mu\}) \cup \{\eta +\!\!\!+\, \mu \mid \mu \in \mathscr{D}\pi\} \\ \tau'\nu &= \begin{cases} \pi\mu, & \text{if } \nu = \eta +\!\!\!+\, \mu \text{ (for some } \mu) \\ \tau\nu, & \text{otherwise.} \end{cases} \end{aligned}$$

As previously noted, I will simply write $s\ \tau_1\ \ldots\ \tau_n$ for the construction of a tree τ with

$$\begin{aligned} \mathscr{D}\tau &= \{\langle\rangle\} \cup \{\langle i\rangle +\!\!\!+\, \eta \mid \eta \in \mathscr{D}\tau_i\} \\ \tau\,\langle\rangle &= s \\ \tau(\langle i\rangle +\!\!\!+\, \eta) &= \tau_i\eta\ . \end{aligned}$$

A.5 Terms

A *term* is a tree over some set of symbols S with fixed *arity*, meaning that the label of a node dictates the number of children of that node. More precisely, the set S is implicitly associated with a function $\mathscr{A} \in S \Rightarrow \mathbb{N}$ defining the arity of each symbol, and, if t is a term, it must be case that

$$t\eta = s \quad \Rightarrow \quad \mathscr{A}s = \max\{i \mid \eta +\!\!\!+\, \langle i\rangle \in \mathscr{D}t\}\ .$$

A *substitution* is the replacement of a 0-ary symbol by another term. If $x \in S$ such that $\mathscr{A}x = 0$ and t and u are terms, then I will (by abuse of notation) let $t[x{:=}u]$ denote the substituion of u for x in t, formally defined as

$$t[x{:=}u] \overset{\text{def}}{=} ((t[\eta_1{:=}u])\cdots)[\eta_n{:=}u] \text{ where } \{\eta^n\} = \{\eta \mid t\eta = x\}\ .$$

As usual, it is convenient to let substitutions be function from terms to terms in their own right, and write $\theta = [x_1{:=}t_1, \ldots, x_n{:=}t_n]$ provided x^n are pairwise distinct, which allows me to write $t\theta$ for the substitution

$$((t[x_1{:=}t_1])\cdots)[x_1{:=}t_1]\ ,$$

as is the usual convention.

Bibliography

[1] Abramov, S. M., and Glück, R. The universal resolving algorithm: inverse computation in a functional language. In *Mathematics of Program Construction. Proceedings* (2000), R. Backhouse and J. N. Oliveira, Eds., vol. 1837 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 187–212.

[2] ACM. *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation* (New York, Sept. 1991), vol. 26(9) of *ACM SIGPLAN Notices*, ACM Press.

[3] Alstrup, S., Secher, J. P., and Thorup, M. Word encoding tree connectivity works. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms* (2000), ACM Press, pp. 498–499.

[4] Augustsson, L. A pattern matching compiler. In *Proceedings of a Workshop on Programs as Data Objects* (Oct. 1986), H. Ganzinger and N. D. Jones, Eds., vol. 217 of *Lecture Notes in Computer Science*, Springer-Verlag.

[5] Bauderon, M., and Courcelle, B. Graph expressions and graph rewritings. *Mathematical Systems Theory 20*, 2–3 (1987), 83–127.

[6] Bauer, F. L., and Wossner, H. *Algorithmic Language and Program Development*. Springer-Verlag, 1982.

[7] Bird, R. S. Tabulation techniques for recursive programs. *ACM Computing Surveys 12*, 4 (Dec. 1980), 403–417.

[8] Bjørner, D., Ershov, A. P., and Jones, N. D., Eds. *Partial Evaluation and Mixed Computation* (Amsterdam: North-Holland, 1988), Elsevier Science Publishers B.V.

[9] Bol, R. Loop checking in partial deduction. *Journal of Logic Programming 16*, 1&2 (1993), 25–46.

[10] Bollig, B., and Wegener, I. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers 45*, 9 (1996), 993–1002.

[11] Boquist, U. *Code Optimization Techniques for Lazy Functional Languages.* PhD thesis, Department of Computer Science, Chalmers University of Technology, Sweden, Mar. 1999.

[12] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers 35*, 8 (Aug. 1986), 677–691.

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

[14] Chin, W. N. *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, Mar. 1990.

[15] Chin, W.-N. Safe fusion of functional expressions. In *Proceedings of the 1992 Conference on LISP and Functional Programming* (San Francisco, June 22–24, 1992), ACM SIGPLAN, SIGACT, and SIGART, pp. 11–20.

[16] Chin, W.-N. Towards an automated tupling strategy. In *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation* (Copenhagen, Denmark, 14-16 June 1993), ACM Press, pp. 119–132.

[17] Chin, W.-N. Fusion and tupling transformations: Synergies and conflicts. In *Fuji International Workshop on Functional and Logic Programming* (Fuji Susono, Japan, July 1995), World Scientific, pp. 176–195. Invited Paper.

[18] Chin, W.-N., and Darlington, J. A higher-order removal method. *Lisp and Symbolic Computation 9*, 4 (Dec. 1996), 287–322.

[19] Chitil, O. Type inference builds a short cut to deforestation. In *Proceedings 4th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'99, Paris, France, 27–29 Sept 1999* (New York, Sept. 27–29 1999), vol. 34(9) of *ACM SIGPLAN Notices*, ACM Press, pp. 249–260.

[20] Comon, H., and Lescanne, P. Equational problems and disunification. *Journal of Symbolic Computation 7*, 3–4 (Mar.–Apr. 1989), 371–425.

[21] Consel, C., and Danvy, O. Partial evaluation of pattern matching in strings. *Information Processing Letters 30*, 2 (1989), 79–86.

[22] Consel, C., and Danvy, O. Tutorial notes on partial evaluation. In *Proceedings, POPL'83* (New York, NY, USA, 1993), ACM, ACM Press, pp. 493–501.

[23] Corbin, J., and Bidoit, M. A rehabilitation of Robinson's unification algorithm. *Information Processing* (1983), 909–914.

[24] Danvy, O., Glück, R., and Thiemann, P., Eds. *Partial Evaluation*, vol. 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[25] de Moor, O., and Secher, J. P. Common-subexpression elimination of conditional expressions. submitted for publication, June 2001.

[26] De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming 41*, 2–3 (1999), 231–277.

[27] Drechsler, R., and amd Bernd Becker, N. G. Learning heuristics for OBDD minimization by evolutionary algorithms. In *Parallel Problem Solving from Nature – PPSN IV* (Berlin, 1996), H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, Eds., Springer, pp. 730–739.

[28] Faxén, K.-F. Optimizing lazy functional programs using flow-inference. In *Workshop on Types for Program Analysis* (Aarhus, Denmark, May 1995), Nielson and Solberg, Eds., Aarhus University, pp. 31–47.

[29] Ferguson, A., and Wadler, P. When will deforestation stop? In *Glasgow Workshop on Functional Programming* (1988), pp. 39–56.

[30] Friedman, S. J., and Supowit, K. J. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers 39*, 5 (May 1990), 710–713.

[31] Futamura, Y. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls 2*, 5 (1971), 45–50.

[32] Futamura, Y., and Nogi, K. Generalized partial computation. In Bjørner et al. [8], pp. 133–151.

[33] Futamura, Y., and Nogi, K. Generalized partial computation. In Bjørner et al. [8], pp. 133–151.

[34] Gallagher, J. Tutorial in specialisation of logic programs. In *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation* (1993), ACM Press, pp. 88–98.

[35] Gill, A., Launchbury, J., and Peyton Jones, S. L. A short cut to deforestation. In *Proceedings, FPCA'93* (Copenhagen, Denmark, June 9–11, Apr. 1993), ACM, ACM Press, pp. 223–232.

[36] Glück, R., and Klimov, A. V. Occam's razor in metacomputation: the notion of a perfect process tree. In *Static Analysis. Proceedings* (1993), G. Filè, P.Cousot, M.Falaschi, and A. Rauzy, Eds., vol. 724 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 112–123.

[37] Glück, R., and Klimov, A. V. Occam's razor in metacomputation: the notion of a perfect process tree. In *Workshop on Static Analysis* (1993), P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, Eds., vol. 724 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 112–123.

[38] Glück, R., and Sørensen, M. A roadmap to metacomputation by supercompilation. In Danvy et al. [24], pp. 137–160.

[39] Gustavsson, J., and Sands, D. A foundation for space-safe transformations of call-by-need programs. In *Electronic Notes in Theoretical*

Computer Science (2000), A. Gordon and A. Pitts, Eds., vol. 26, Elsevier Science Publishers.

[40] Hickey, T., and Smith, D. Toward the partial evaluation of CLP languages. In PEPM'91 [2], pp. 43–51.

[41] Higman, G. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society (3) 2* (1952), 326–336.

[42] Hindley, J. R. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc 146* (1969), 29–60.

[43] Hoffmann, B., and Plump, D. Implementing term rewriting by jungle evaluation. *RAIRO Theoretical Informatics and Applications 25(5)* (1991), 445–472.

[44] Hughes, J. Super combinators - A new implementation method for applicative languages. In *Proceedings, Symposium on Lisp and Functional Programming*. Association for Computing Machinery, 1982, pp. 1–10.

[45] Johnsson, T. *Compiling Lazy Functinoal Languages*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Sweden, Feb. 1987.

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

[47] Jones, N., Sestoft, P., and Søndergaard, H. An experiment in partial evaluation: the generation of a compiler generator. In *Rewriting Techniques and Applications* (1985), J.-P. Jouannaud, Ed., vol. 202 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 124–140.

[48] Jones, N., Sestoft, P., and Søndergaard, H. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation 2*, 1 (1989.), 9–50.

[49] Jones, N. D. An introduction to partial evaluation. *ACM Computing Surveys 28*, 3 (Sept. 1996), 480–504.

[50] JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[51] JONES, S. L. P. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming 2*, 2 (July 1992), 127–202.

[52] KAHRS, S. Unlimp – uniqueness as a leitmotiv for implementation. In *Programming Language Implementation and Logic Programming* (August 1992), M. Bruynooghe and M. Wirsing, Eds., vol. 631 of *Lecture Notes in Computer Science*, Springer, pp. 115–129.

[53] KIRCHNER, C., AND LESCANNE, P. Solving disequations. In *Proceedings, LICS'87* (Ithaca, New York, 22–25 June 1987), The Computer Society of the IEEE, pp. 347–352.

[54] KLEENE, S. C. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[55] KLOP, J. W. Term rewriting systems. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds., vol. 2. Oxford University Press, Oxford, 1992, ch. 1, pp. 1–117.

[56] KÜHNEMANN, A. Benefits of tree transducers for optimizing functional programs. In *Partial Evaluation and Program Transformation* (1999), R. Gück and Y. Futamura, Eds., Waseda University, pp. 61–82.

[57] KÜHNEMANN, A. Comparison of deforestation techniques for functional programs and for tree transducers. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming* (Tsukuba, Japan, 1999), A. Middeldorp and T. Sato, Eds., vol. 1722, Springer-Verlag, pp. 114–130.

[58] LAFAVE, L., AND GALLAGHER, J. P. Partial evaluation of functional logic programs in rewriting-based languages. Tech. Rep. CSTR-97-001, Department of Computer Science, University of Bristol, Mar. 1997.

[59] LAFAVE, L., AND GALLAGHER, J. P. Extending the power of automatic constraint-based partial evaluators. *ACM Computing Surveys 30*, 3es (Sept. 1998). Article 15.

[60] Lamping, J. An algorithm for optimal lambda calculus reduction. In *POPL '90. Proceedings of the seventeenth annual ACM symposium on Principles of programming languages, January 17–19, 1990, San Francisco, CA* (New York, NY, USA, Jan. 1990), P. Hudak, Ed., ACM Press, pp. 16–30.

[61] Lassez, J.-L., Maher, M., and Marriott, K. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, Los Altos, Ca., 1988, pp. 587–625.

[62] Launchbury, J., and Sheard, T. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)* (La Jolla, California, June 25–28, 1995), ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press, pp. 314–323.

[63] Lee, C. S., Jones, N. D., and Ben-Amram, A. M. The size-change principle for program termination. In *POPL 2001. Proceedings of the 28th ACM SIGPLAN-SIGACT on Principles of programming languages* (New York, NY, USA, Jan. 2001), ACM, Ed., ACM Press, pp. 81–92.

[64] Leuschel, M. A formal comparison of well-quasi and well-founded orders for online termination. In *International Workshop on Logic-based Program Synthesis and Transformation* (1998).

[65] Leuschel, M., and De Schreye, D. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing* (1997).

[66] Leuschel, M., Martens, B., and Schreye, D. D. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems 20*, 1 (Jan. 1998), 208–258.

[67] Leuschel, M., and Sørensen, M. Redundant argument filtering of logic programs. In *Logic Program Synthesis and Transformation* (1996), J. Gallagher, Ed., vol. 1207 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 83–103.

[68] M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12*, 1 (Jan. 1993), 6–12.

[69] Mendelson, E. *Introduction to Mathematical Logic*, 3rd ed. Wadsworth Mathematical Series. Wadsworth &amp; Brooks / Cole Advanced Books &amp; Software, Monterey, CA, 1987. (1st ed., D. Van Nostrand Co., Princeton, NJ, 1964; 2nd ed., D. Van Nostrand Co., New York, 1979).

[70] Michie, D. "memo" functions and machine learning. *Nature* (Apr. 1968), 19–22.

[71] Milner, R. A theory of type polymorphism in programming languages. *Journal of Computer and System Science 17*, 3 (1978), 348–375.

[72] Møller, J., Andersen, H. R., and Hulgaard, H. Product configuration over the internet. In *Proceedings 6th International INFORMS Conference on Information Systems and Technology* (Miami Beach, Florida, Nov. 2001).

[73] Mycroft, A. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming* (Toulouse, France, Apr. 1984), M. Paul and B. Robinet, Eds., vol. 167 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 217–228.

[74] Necula, G. C. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Sept. 1998.

[75] Nelson, G., and Oppen, D. C. Fast decision procedures based on congruence closure. *Journal of the ACM 27*, 2 (Apr. 1980), 356–364.

[76] O'Donnell, M. J. *Computing in systems described by equations*, vol. 58 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1977. Programming languages (Electronic computers).

[77] Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.

[78] Partsch, A. *Specification and Transformation of Programs—A Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[79] Paterson, M., and Wegman, M. Linear unification. *Journal of Computer and System Sciences 16* (1978), 158–167.

[80] Pettorossi, A. *Methodologies for transformations and memoing in applicative languages.* PhD thesis, Department of Computer Science, University of Edinburgh, Scotland, 1984. CST-29-84.

[81] Pettorossi, A. A powerful strategy for deriving efficient programs by transformation. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Aug. 1984), ACM, ACM, pp. 273–281.

[82] Pettorossi, A., Pietropoli, E., and Proietti, M. The use of the tupling strategy in the development of parallel programs. In *Parallel Algorithm Derivation and Program Transformation*, R. Paige, J. Reif, and R. Wachter, Eds. Kluwer Academic Publishers, 1993, pp. 111–151.

[83] Pettorossi, A., and Proietti, M. Rules and strategies for program transformation. In *Formal Program Development, IFIP TC2/WG 2.1 State-of-the-Art Report* (July 1993), B. Moeller, H. Partsch, and S. Schuman, Eds., vol. 755 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 263–304.

[84] Pettorossi, A., and Proietti, M. A comparative revisitation of some program transformation techniques. In Danvy et al. [24], pp. 355–385.

[85] Pettorossi, A., and Proietti, M. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys 28*, 2 (June 1996), 360–414.

[86] Plasmeijer, R., and van Eekelen, M. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[87] Reynolds, J. C. An introduction to the polymorphic lambda calculus. In *Logical Foundations of Functional Programming*, G. Huet, Ed., Uni-

versity of Texas at Austin Year of Programming Series. Addison-Wesley, Reading, MA, 1990, pp. 77–86.

[88] RICHARD RUDELL. Dynamic variable ordering for ordered binary decision diagrams. In *IEEE /ACM International Conference on CAD* (Santa Clara, California, Nov. 1993), ACM/IEEE, IEEE Computer Society Press, pp. 42–47.

[89] ROMANENKO, A. The generation of inverse functions in Refal. In Bjørner et al. [8], pp. 427–444.

[90] ROMANENKO, A. Inversion and metacomputation. In PEPM'91 [2], pp. 12–22.

[91] SABIN, D., AND WEIGEL, R. Product configuration frameworks—a survey. *Journal of the IEEE Intelligent Systems & their applications* (jul 1998), 42–49.

[92] SAHLIN, D. Mixtus: An automatic partial evaluator for full prolog. *New Generation Computing 12*, 1 (1993), 7–51.

[93] SASSA, M., AND GOTO, E. A hashing method for fast set operations. *Information Processing Letters 5*, 2 (June 1976), 31–34.

[94] SECHER, J. P. Perfect supercompilation. Tech. Rep. DIKU-TR-99/1, Department of Computer Science (DIKU), University of Copenhagen, Feb. 1999.

[95] SECHER, J. P. Driving in the jungle. In *Proceedings of the Second Symposium on Programs as Data Objects* (May 2001), O. Danvy and A. Filinski, Eds., vol. 2053 of *Lecture Notes in Computer Science*, BRICS, Springer-Verlag, pp. 198–217.

[96] SECHER, J. P., AND SØRENSEN, M. H. From checking to inference via driving and dag grammars. In *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation* (Jan. 2002), P. Thiemann, Ed., ACM Press, ACM Press, pp. 41–51.

[97] SECHER, J. P., AND SØRENSEN, M. H. B. On perfect supercompilation. In *Proceedings of Perspectives of System Informatics* (2000),

D. Bjørner, M. Broy, and A. Zamulin, Eds., vol. 1755 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 113–127.

[98] Seidl, H., and Sørensen, M. Constraints to stop deforestation. *Science of Computer Programming 32* (1998), 73–107.

[99] Sestoft, P. Deriving a lazy abstract machine. *Journal of Functional Programming 7*, 3 (May 1997), 231–264.

[100] Sheeran, M., and Stålmarck, G. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design: An International Journal 16*, 1 (Jan. 2000), 23–58.

[101] Sleator, D. D., and Tarjan, R. E. A data structure for dynamic trees. *Journal of Computer and System Sciences 26*, 3 (1983), 362–391.

[102] Smith, D. Partial evaluation of pattern matching in constraint logic programming. In PEPM'91 [2], pp. 62–71.

[103] Sørensen, M. Turchin's supercompiler revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.

[104] Sørensen, M., and Glück, R. An algorithm of generalization in positive supercompilation. In *Logic Programming: Proceedings of the 1995 International Symposium* (1995), J. Lloyd, Ed., MIT Press, pp. 465–479.

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

[106] Sørensen, M. H., and Glück, R. Introduction to supercompilation. In *Partial Evaluation: Practice and Theory* (1999), J. Hatcliff, T. Mogensen, and P. Thiemann, Eds., vol. 1706 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 246–270.

[107] Sørensen, M. H., and Secher, J. P. From type inference to configuration. submitted for publication, May 2002.

[108] Sørensen, M. H. B. Convergence of program transformers in the metric space of trees. *Science of Computer Programming 37*, 1–3 (May 2000), 163–205.

[109] Takano, A. Generalized partial computation for a lazy functional language. In PEPM'91 [2], pp. 1–11.

[110] Takano, A. Generalized partial computation using disunification to solve constraints. In *Conditional Term Rewriting Systems. Proceedings* (1993), M. Rusinowitch and J. Remy, Eds., vol. 656 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 424–428.

[111] Takano, A., and Meijer, E. Shortcut deforestation in calculational form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)* (La Jolla, California, June 25–28, 1995), ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press, pp. 306–313.

[112] Tamaki, H., and Sato, T. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Conference on Logic Programming* (Uppsala, 1984), S.-Å. Tärnlund, Ed., pp. 127–138.

[113] Tarjan, R. E. Efficiency of a good but not linear set merging algorithm. *Journal of the ACM 22* (1975), 215–225.

[114] Turchin, V. Semantic definitions in Refal and the automatic production of compilers. In *Workshop on Semantics-Directed Compiler Generation, Århus, Denmark* (1980), N. Jones, Ed., vol. 94 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 441–474.

[115] Turchin, V. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems 8*, 3 (1986), 292–325.

[116] Turchin, V. Metacomputation: Metasystem transition plus Supercompilation. In Danvy et al. [24], pp. 481–510.

[117] Turchin, V., and Nemytykh, A. A self-applicable supercompiler. Technical Report CSc. TR 95-010, City College of the City University of New York, 1995.

[118] Turchin, V. F. The algorithm of generalization in the supercompiler. In Bjørner et al. [8], pp. 531–549.

[119] Veron, M., Fargier, H., and Aldanondo, M. From CSP to configuration problems. In *Configuration: Papers from the AAAI workshop*, B. Faltings, E. C. Freuder, G. Friedrich, and A. Felfernig, Eds. The AAAI Press, 1999, pp. 101–106. Available at http://wwwold.ifi.uni-klu.ac.at/Conferences/aaai99_ws_configuration.

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

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

[122] Wadsworth, C. P. *Semantics and pragmatics of the lambda calculus*. Ph.D. thesis, Programming Research Group, Oxford University, Sept. 1971.

[123] Wegener, I. *The Complexity of Boolean Functions*. Wiley Teubner Series in Computer Science. John Wiley and Sons, New York, 1987.

Index

Symbols