To my parents

Acknowledgements

First of all, I would like to thank my advisor Peter Fritzson for coming up with the idea of using algorithmic debugging for lazy functional languages in the first place, thereby initiating this work, and for co-authoring several papers on the topic. Peter has also been very approachable and generally helpful, which I really appreciate. I would also like to express my gratitude to Mariam Kamkar and Jan Maluszynski, the two other members of my advisory group, for their support and many insightful pieces of advice.

Then I would like to thank my other friends and colleagues at PELAB for being around and creating a nice and friendly atmosphere. In particular, I would like to thank Lars Viklund, Mikael Petterson and Niclas Andersson for always taking time to answer innumerable technical questions and generally helping me out, not the least with various LaTeXproblems.

I would also like to thank Jan Sparud and Thomas Johnsson, Chalmers, and Roger Skagervall, Carlstedt Elektronik AB, for interesting discussions and suggestions.

Finally I would like to thank our technical and administrative staff, and then in particular Lillemor Wallgren and Gunilla Blom-Lingenhult for taking care of us.

Special thanks to Jonas S Karlsson for many enjoyable duels, of all kinds.

Thanks a lot, all of you!

This work has been supported by The Swedish Board for Industrial and Technical Development (NUTEK).

Contents

Chapter 1

Introduction

For various reasons, as will be explained in the following, debugging programs written in lazy functional languages is difficult. There are currently no realistic, general purpose debugging tools available, confining lazy functional programmers to use only the most primitive of debugging techniques, e.g. bottom-up testing of individual functions. Clearly, the situation is rather unsatisfying, and it is imperative that something is done about it, if lazy functional languages are to gain popularity and credibility on a larger scale outside a dedicated scientific community.

The aim of our work is to provide a realistic basis for debugging tools for lazy functional languages. At the current state of our work, we cannot claim to have done so, but it is our hope that the techniques suggested in this thesis will scale up to 'real world' problems. From our initial experience, we think that there is a good chance that they will.

1.1 Motivation

Functional programming has received widespread and increasing interest in both the academic world and in the industry during the last ten to fifteen years or so. The reasons for this are many and diverse, see for example the excellent survey paper by Hughes [Hug89] or the one by Hudak [Hud89]. To mention two, modern (pure[1]) functional languages are very expressive, leading to programs that are concise as well as easy to understand, and, because of referential transparency, it is easier to formally manipulate and reason about a functional program than about a program written in an imperative language. Indeed, it is often possible to read and understand a functional program *declaratively*, which is why functional programming languages, together with logic programming languages, sometimes are re-

[1] I.e. free from updateable variables (side-effects) at the language level. Strictly speaking, to qualify as functional, a language must be pure. However, many almost functional languages, such as ML, are often referred to as functional anyway. Lazy functional languages, which are the ones considered in this thesis, are on the other hand almost invariably pure because imperative assignment and laziness do not mix.

ferred to as *declarative languages*. Now, while one should be very careful about using the term declarative in any absolute sense, it is safe to say that functional languages are more declarative than are imperative ones.

Another often stated advantage of functional programming is the comparative ease by which such programs may be efficiently executed in parallel, even on massively parallel computers, without having to change the programs, at least not in any substantial way (see e.g. [PJ87, pp. 409–430] or [Szy91]). This in one of the motivations behind the ongoing, large scale industrial hardware and software development project at Carlstedt Elektronik AB, aiming at declarative (currently functional) programming for embedded real-time systems [LSB94].

The class of functional languages may be divided into two subclasses: *strict functional languages*[2], employing call-by-value semantics, and *non-strict functional languages*, frequently (though somewhat incorrectly) also referred to as *lazy functional languages*, employing call-by-need semantics. While both kinds on the whole share many advantages (and disadvantages) when compared to more traditional, imperative languages, both do have their own relative merits and weaknesses.

This is not, however, the place for an extensive discussion of lazy versus strict functional languages. We will just contend ourselves by noting that lazy functional languages are used in practice, both by academics and by people in industry, and that a particular problem related to this class of languages concerns debugging, thus motivating the work presented in this thesis. Moreover, to spread the use of lazy functional languages more widely, suitable debugging tools will be necessary.

The lack of debugging tools has been apparent for quite some time, and, as has been pointed out earlier by others [HO85, OH88, Toy87], it is clear that traditional debugging techniques (e.g. breakpoints, tracing, variable watching etc. in the conventional sense) are not particularly well suited in this context because computations in a lazy program, due to the demand driven execution model, generally do not take place in the order one might expect from reading the source code. Moreover, many programming errors common in imperative languages such as C, e.g. storage related ones such as dangling or uninitialized pointers, just do not occur in a lazy functional language, further motivating specialized tools and techniques. On the other hand it is equally clear that debugging tools *are* needed, despite functional languages being 'declarative', since there is always the possibility that a program does not express its author's intentions, even when regarded as a specification.

1.2 A Possible Solution

So, how should one go about debugging lazy functional programs? Well, functional languages being more declarative than imperative ones, it seems reasonable to use more declarative debugging techniques for them. To put

[2] Also known as *eager* or *applicative* languages.

it another way, the abstraction level during debugging should match the abstraction level during programming. In particular, the user should not have to concern herself with language implementation details such as evaluation order. After all, one of the benefits of writing declarative programs in a declarative language is that it allows us to (largely) forget about such operational issues.

Algorithmic debugging, introduced by Shapiro [Sha82], is one such technique. Debugging takes place at the procedure abstraction level by querying the user about the procedure invocations that took place during execution: 'Should this procedure, given these arguments, yield the result it did?' Given correct answers to its questions, the debugger will locate the erroneous procedure.

The order in which the questions are asked, which usually reflects the order in which the procedures were invoked, is of no importance: the user concentrates on one question at a time, and answers it according to how the procedure in question should work, with reference to some high level, formal or informal, specification. Hence algorithmic debugging may readily be used in a lazy functional context, the order independence and the declarative flavour of the technique being our original reasons for trying to use algorithmic debugging to this end.

Briefly, algorithmic debugging is a two phase process. An *execution tree*, i.e. a tree-structured trace, is built at the procedure abstraction level during the first phase, the *trace phase*, during the execution of the program. This tree is then traversed during the second phase, the *debugging phase*. Each node in the tree corresponds to an invocation of a procedure or function and holds a record of actual arguments and returned results. Its children corresponds to the procedure or function invocations that took place during the invocation in question.

Once built, the debugger typically traverses the tree in a top-down manner, asking, for each encountered node, whether the recorded procedure or function invocation is correct or not. If not, the debugger will continue with the children of the node, otherwise with its siblings. A bug has been found when an erroneous node is identified where all children (if any) behaved correctly.

Obviously, there must be an *externally visible bug symptom* for this technique to work: e.g. if two bugs conspire to hide each other, it will not be possible to (algorithmically) debug the program. But if there is no bug symptom, the user is unlikely to undertake any debugging in the first place, so this is not really any major drawback.

1.3 Theory Put into Practice

To test the idea in practice, support for algorithmic debugging was added to an existing compiler for a small lazy functional language, and an algorithmic debugger, the Lazy Algorithmic Debugging Tool (LADT), was implemented. The language, called Freja [Nil91], is essentially a subset of

Miranda[3]. It is based on graph reduction and implemented using a G-machine approach [Aug87, Joh87, PJ87]. LADT was successfully applied to a number of small examples, thus showing the relevance of algorithmic debugging for lazy functional programming.

As we gained experience from using this early system, a number of problems became apparent. First, the questions asked were often big and complex, involving large data structures and intermediate results in the form of unevaluated expressions in argument and result positions. Second, storing the complete trace is impractical for any but the smallest of problems. Finally, the number of questions asked by the debugger for realistic programs were too large.

To alleviate the first problem, a technique which we term *strictification* was introduced. This technique is concerned with hiding the lazy evaluation order by restructuring the execution tree to make it more like a tree resulting from tracing in a strict language, hence the name.

It may seem surprising that the lazy evaluation order has to be hidden, since it was stated above that the order was of no importance when doing algorithmic debugging. But to successfully abstract away from evaluation order details, it is, in the case of lazy functional languages, not enough to be able to disregard the order in which functions are invoked. The lazy evaluation strategy manifests itself all too well due to the presence of the above mentioned intermediate results, that neither directly correspond to constructs in the source code, nor in any obvious way to the values they represent and in terms of which the user probably reasons about the program.

By performing strictification, the number of unevaluated expressions involved in questions posed to the user is substantially reduced, which, in our opinion, tends to make them easier to understand and answer. Hence, some kind of strictification or similar technique to obtain a strictified (or *applicative*) trace, must be regarded as an essential abstraction step if the details of the underlying lazy evaluation machinery are really to be hidden. Further, this does not only apply to algorithmic debugging, but to any trace based debugging technique when used in a lazy context.

Note that it is not possible to simply evaluate unevaluated expressions as they are encountered during the debugging phase, since they may represent infinite structures or non-terminating computations. In addition, if this is done, algorithmic debugging may in the general case fail to locate the bug because computations that are not reflected in the execution tree take place before the question is posed to the user.

Our current implementation of LADT thus does strictification which we have found to be very useful. This leaves the last two, rather severe, problems. We do not believe that an algorithmic debugger will be practically usable unless these are addressed in a satisfactory manner. Moreover, doing strictification afterwards is less than ideal for a number of reasons. It would be much better if a strictified trace could be built directly. It would be even better if only as much of the trace as there is room for was to be

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

stored at any one time, other parts of the trace being created on demand by re-executing the program, thus addressing the trace storage problem. Techniques for achieving both objectives, *immediate tree construction* and *piecemeal tracing*, are outlined in this thesis, and we believe this to be one of its main contributions, laying a foundation for a realistic 'lazy' debugger.

1.4 Thesis Overview

The rest of the thesis is organised as follows. Chapter 2 gives some technical background on functional languages, their implementation and algorithmic debugging. Readers familiar with any of these areas may readily skip the corresponding sections.

Chapter 3 explains the special problems related to debugging for lazy functional languages and gives an overview of other work in the area. Some conclusions are drawn from this, motivating the basic approach used in LADT.

LADT is then described in chapter 4. The used techniques are explained and some implementation details are given. LADT is also assessed and some of its main weaknesses are pointed out and explained, motivating the ideas presented in the following chapters.

Chapter 5 discusses requirements for a realistic debugger and outlines a number of ideas for meeting these. Among others, it elaborates on the refined trace technique alluded to above. An idea concerning the problem of debugging so called list comprehensions in an algorithmic framework is also presented. The latter is a particular instance of the problem of allowing the user to debug at the source code level.

Plans for the future are summarized in chapter 6 and conclusions are finally given in chapter 7. In the appendices, the syntax of Freja and a larger debugging example are presented. Also, a small demonstration program to illustrate the proposed method for debugging list comprehensions is provided.

Chapters 3 and 4 are more or less directly taken from from Nilsson & Fritzson [NF94] (which in turn is based on [NF92] and [NF93]), whereas chapter 5 partly builds on ideas presented in that paper, though further developed, and partly on previously unpublished material.

Chapter 2

Technical Background

This chapter will try to give some technical background to the rest of this thesis, covering the areas of functional programming, implementation of lazy functional languages, and algorithmic debugging. The aim is to make the thesis self contained, not to treat any of the subjects in any depth. For a comprehensive survey of the functional programming field, the reader is referred to Hudak [Hud89]. For a thorough introduction to functional programming and implementation of functional languages, see Bird & Wadler [BW88], Field & Harrison [FH88] and Peyton Jones [PJ87].

The implementation part focuses on graph rewriting and the G-machine, since this is what the current Freja implementation is based on. However, graph rewriting is the predominating technique for efficient implementation of lazy functional languages, both on stock hardware and on more specialized machines, and since the techniques presented later in this thesis could be adapted to other graph rewriting frameworks, we believe that they have quite wide applicability.

The organisation of this chapter is as follows. First, in section 2.1 various functional programming languages and some common features of them are discussed. Then, in section 2.2, functional programming is discussed in general terms and some examples are given. Section 2.3 addresses implementation of functional languages and finally the ideas behind algorithmic debugging are outlined in section 2.4.

2.1 Functional Programming Languages

Over the years, numerous functional or nearly functional languages have emerged, for example ML [GMW79, Mil84], Hope [BMS80], LML [Aug84], Miranda [Tur85] and Haskell [HPJW+92]. These and other languages have all contributed to the set of features that are now thought to characterize a modern functional language, of which some of the most important ones are:

- Functions are first class objects.

- Strong, polymorphic type system.
- Pattern matching.

Taken together, these features yield languages of remarkable power and expressiveness, and we will discuss them a bit further in the next section.

Of the languages mentioned above, the last three are lazy whereas ML and Hope are strict, although Hope has some lazy features. (Different evaluation orders and laziness are discussed in section 2.3.3.) ML also has imperative constructs, and is thus not quite a functional language, as discussed below. Of the three lazy ones, Haskell, being the most recent, in many ways represents the current state of the art.

This thesis is concerned with lazy languages, so the techniques that will be presented could typically be used in Miranda and Haskell implementations. However, the language used in this thesis will mainly be Freja [Nil91], since the work so far has been carried out in the context of this language. Freja is basically a subset of Miranda, but its syntax is not that far from that of Haskell, or other similar languages, either.

Functional languages are nice in many ways, but there are also some not so nice aspects of them. In particular, it is rather difficult to build really efficient language implementations, and lazy functional languages tend to be worse in this respect than strict ones. For a long time, it seemed as if the basic computational model of functional languages, especially lazy ones, was too far removed from the underlying von Neumann architecture of ordinary computers to permit efficient implementations. This led to attempts to build special functional computers.

While specialized, massively parallel functional machines might be the way to go in the future, it it is obviously very desirable to have efficient implementations of this class of languages on standard hardware. In their G-machine papers, Augustsson [Aug84] and Johnsson [Joh84] showed that reasonably efficient implementations of lazy functional languages are indeed possible on stock hardware. Hudak & Kranz [HK84] attained similar results. The basic idea is to transform all user functions into combinators or 'rewrite rules'. These rules are then compiled into fixed, fast code sequences that perform the corresponding rewrites when executed. This is discussed further in the implementation section below.

2.2 Functional Programming

A functional program consists of a set of function and constant definitions. All computations take place by applying functions to arguments. There is no explicit state being transformed by executing a sequence of commands, and in particular there are no updateable variables and no side-effects. Conceptually, the only way to bind a variable to a value is by means of function application, by which a formal parameter is bound to an actual parameter, even though there might in practice be other syntactic binding constructs for the convenience of the user or for efficiency reasons.

This means that once a variable has been bound, there is no way its value can be changed, i.e. it is not really varying at all! Functions in a functional language thus become functions in the mathematical sense, in that they will always yield the same result when applied to some particular value. The result of a function application may therefore always be substituted for any instance of the application. The result of the computation is guaranteed to stay the same. This property is called *referential transparency*, and because of it, it is comparatively easy to formally reason about and manipulate functional programs, an often stated advantage of functional languages.

Compare this with the situation in an imperative language where a 'function' may return a different result each time it is called. As a case in point, consider the C function `getc` that returns the next character from the file specified by its argument.

Now, there are functional languages in the sense above (sometimes called *pure* languages to emphasize that they are referentially transparent) and there are almost functional languages, that, while having a very functional flavour, do support imperative, updateable variables as well (e.g. ML). This thesis is however only concerned with pure languages, since lazy functional languages generally are pure (imperative variables and laziness do not go very well together) and since referential transparency makes it a lot easier to apply algorithmic debugging.

Another distinguishing feature of functional languages is that functions are first class objects, i.e. they can be passed as arguments, returned as results or built into data structures, just like any other type of object such as integers or lists. A function that takes another function as an argument, or returns a function as its result, is called *higher order*. A typical example of this is the function `map` that takes a function and a list as its arguments, applies the function to each element in the list and returns a new list containing the results of the applications.

Considerable effort has been spent on providing rigid mathematical foundations for functional programming languages. Examples of this are domain theory, lambda calculus, and type system theory. However, with the exception of the lambda calculus, we will not, because of the practical nature of this thesis, concern ourselves further with the underlying theory. A nice little introduction to domain theory can be found in Field & Harrison [FH88, pp. 547–565], and a more thorough treatment in Stoy [Sto77]. See Cardelli & Wegner [CW85] for a good account on type systems.

We will conclude this section with three small program examples to introduce the syntax of Freja by way of example (an EBNF syntax specification may be found in appendix A), and to give the reader who is unfamiliar with the functional programming paradigm an idea of what a simple functional program might look like. The first program calculates the factorial of 5:

```
fac 0       = 1;
fac (n + 1) = (n + 1) * (fac n);

main = fac 5;
```

The special constant `main` is bound to the top level expression which is going to be evaluated to get the result of the program. Note the use of *pattern matching* to define the function `fac`: the first clause only matches when the function is applied to 0, the second clause only when the function is applied to an argument greater than or equal to 1, in which case `n` is bound to the argument minus 1. Also note that the parentheses are left out around the argument, both in the function definitions and when the functions are applied.

The second program defines a function that calculates the length of a list and uses this to find the length of the list `[7, 11, 3, 5]` (which is 4):

```
len [] = 0;
len (x:xs) = 1 + len xs;

main = len [7, 11, 3, 5];
```

Again, note the use of pattern matching. The pattern `[]` denotes the empty list, and the first clause of `len` is therefore only applicable when `len` is applied to empty lists. The pattern `(x:xs)` matches any non-empty list and, in the case of successful matching, will cause `x` to be bound to the first element in the list, its *head*, and `xs` to the rest of the list, its *tail*. The symbol `:` is the `cons`-operator, used to prepend an element to a list.

Finally, here is the definition of the function `map`, as mentioned above, and an example of its use:

```
map f [] = [];
map f (x:xs) = (f x) : (map f xs);

sqr x = x * x;

main = map sqr [1, 2, 3];
```

The result of running this program is the list `[1, 4, 9]`.

In the two last examples, note that nothing was assumed about the *type* of the list elements in the definitions of `len` and `map`, or about the type of the function `f` in the definition of `map`. This means that these functions may be applied to lists containing elements of any (one) type (and, in the case of `map`, to any function mapping objects of the list element type to any other type of object). Modern functional languages often have *strong, polymorphic type systems* [CW85] that permit this kind of definitions, while making sure that functions are used in a consistent way through *type inference* in order that the programs do not go wrong at runtime because of type errors.

2.3 Implementation of Functional Languages

This section gives an introduction to implementation of functional languages with an emphasis on lazy ones. Starting with a brief discussion on

exp	$\rightarrow$	*const*	Built-in constants
	$\mid$	*var*	Variable names
	$\mid$	*exp exp*	Function application
	$\mid$	$\lambda var.\, exp$	Lambda abstractions
const	$\rightarrow$	`0` $\mid$ `1` $\mid$ `2` $\mid \ldots$	Integers
	$\mid$	`+` $\mid$ `-` $\mid$ `*` $\mid \ldots$	Arithmetic functions
	$\mid$	$\ldots$	Other constants
var	$\rightarrow$	`x` $\mid$ `y` $\mid$ `z` $\mid \ldots$	

Figure 2.1: Syntax of the lambda calculus with constants.

lambda calculus and different evaluation strategies, the basic principles of graph reduction and supercombinator compilation will be developed step by step, leading to a description of the G-machine.

2.3.1 The Lambda Calculus

The lambda calculus is fundamental to the theory behind functional languages. Originally developed to give operational semantics to mathematical functions [Chu41], it turned out to be an ideal vehicle for both description and implementation of functional languages. Introductions to the lambda calculus can be found in both Peyton Jones [PJ87, pp. 9–35] and in Field & Harrison [FH88, pp. 111–140]. For a comprehensive treatment of the subject, see e.g. Barendregt [Bar84].

What makes the lambda calculus so attractive, is that it is a very simple language, yet sufficiently powerful to express any computable function, i.e. any program that can be written in any computer language may also be expressed in the lambda calculus. In particular, any functional program may be translated into lambda calculus, and so, given transformation rules for doing this, all such languages can be described in the same, simple framework. Furthermore, the lambda calculus can also serve as a common base for implementation. The syntax of the lambda calculus is given in EBNF in figure 2.1.

In its purest form, the lambda calculus does not even include constants. However, for practical purposes, we are entitled to extend the basic lambda calculus with whatever features we find convenient, as long as we can show how the new constructs can be expressed in the terms of the basic lambda calculus. In the following we will introduce constants as we see fit, e.g. integers and arithmetic functions, taking for granted that they can be expressed in the basic language, since we know that basic lambda calculus is powerful enough to express such functions. But we will in general not be concerned with how this may be done. The interested reader is referred to the literature.

2.3.2 Reducing Lambda Expressions

A *lambda abstraction* (see figure 2.1) is a function of one argument, the variable immediately after the lambda being the formal parameter or *bound variable*, and the part after the dot being the *body* of the lambda abstraction. A *free variable* is an occurrence of a variable that is not bound by any lambda abstraction.

Application of an lambda abstraction to an argument is denoted by juxtaposition, e.g. `F G` means `F` applied to `G`. A lambda abstraction is applied to an argument by replacing the whole of the application with the body of the abstraction, after first having substituted the argument for *free* occurrences of the variable bound by the abstraction in its body. This is known as *β-reduction*.

For an example of bound and free variables, consider the following nested lambda abstractions:

```
(λx.(λy.(λx.z x) y x))
```

Here, `z` occurs free, since it is not bound by any lambda abstraction. All other variable occurrences are bound; the last occurrence of `x` by the outermost lambda abstraction, `y` by the lambda abstraction that constitutes its body, and the other occurrence of `x` by the innermost lambda abstraction. Looking only at the body of the outermost lambda abstraction, `x` occurs both free (the last occurrence) and bound.

Now, to illustrate β-reduction, consider the following lambda expression, consisting of a lambda abstraction applied to an argument:

```
(λx.+ 1 x) 4
```

By β-reduction, this yields:

```
+ 1 4
```

Now, `+`, `1` and `4` are all built-in constants that could be replaced with equivalent lambda expressions to proceed with the reduction. However, for practical reasons, there is a set of rules known as *δ-reductions* defined for the built-in constants. Hence, by δ-reduction we will get the final result which is `5`.

Application is defined to be left-associative, and `+ 1 4` should therefore be interpreted as `(+ 1) 4`, i.e. `+ 1` should be evaluated first and the result of this evaluation should then be applied to `4`. Note that there in priciple is nothing special about arithmetic functions such as `+`. They are treated just like any other lambda abstractions, hence the prefix notation for application.

What then, is the result of applying `+` to `1`? It is simply a new function that adds one to its only argument. This way of treating functions of multiple arguments is known as *currying*, and application of a function to too few arguments is called *partial application*. Call this intermediate function `add1`. The full reduction sequence then becomes:

$$(\lambda\texttt{x.+ 1 x) 4} \stackrel{\beta}{\Rightarrow} \texttt{+ 1 4} \stackrel{\delta}{\Rightarrow} \texttt{add1 4} \stackrel{\delta}{\Rightarrow} \texttt{5}$$

where $\Rightarrow$ means 'reduces to'. Here, the arrows have been labelled by β and δ for clarity.

However, there is no need to do any reduction to get from `(+ 1)` to `add1`. We can simply keep `(+ 1)` as it is, provided that we treat it as a function. Indeed, it turns out that it is very important to perform partial application by accumulating arguments until all arguments are available, and only then do the final reduction, since this means that a precompiled code sequence can be called to perform the reduction (see section 2.3.8).

There are two further rules defined in the lambda calculus: *α-conversion* and *η-reduction*. By α-conversion, the name of a variable bound by a lambda abstraction may be changed, as long as it is done consistently, e.g.:

$$(\lambda\texttt{x.+ x 1}) \stackrel{\alpha}{\Leftrightarrow} (\lambda\texttt{y.+ y 1})$$

where the double arrow indicates that that the rule works both ways. η-reduction can be used to remove redundant lambda abstractions. For example, by η-reduction, we have

$$(\lambda\texttt{x.F x}) \stackrel{\eta}{\Rightarrow} \texttt{F}$$

provided `F` denotes a function in which `x` does not occur free.

The reduction rules may also be applied backwards, and are then known as β-, δ-, and *η-abstraction*, respectively. If the rules are applied both ways, we talk about *conversion*. See the references on lambda calculus for details.

In the following, we will often just talk about reduction, denoted by an arrow, $\Rightarrow$, without explicitly stating which rule that is employed, or how many reduction steps that are involved.

2.3.3 Reduction Orders and Lazy Evaluation

Consider the following lambda expression:

```
if (= 1 1) (* 2 5) (/ 2 0)
```

Here, `if` is a function of three arguments. If the first evaluates to `True`, the expression as a whole will reduce to (the value of) the second argument, otherwise it will reduce to the third argument.

Clearly, there are several ways to reduce this expression. We could, for example, start by reducing `= 1 1` to True, or we could start by reducing `* 2 5` to `10`. Each subexpression that can be reduced is known as a *reducible expression*, or *redex* for short. It is also clear that if we choose a sensible reduction order, the whole expression would eventually reduce to `10`, but if we during our reductions attempted to calculate `/ 2 0` the whole computation would fail.

Fortunately it can be proved that if there is some reduction order that terminates for a given lambda expression, then *normal-order* reduction terminates (and the result will be the same). This is a consequence of the Church-Rosser theorems I and II. (See e.g. Barendregt [Bar84].)

Normal-order reduction states that the outermost, leftmost redex should be reduced first. This effectively implements *call-by-need* or *non-strict* semantics. In our case, this means that `/ 2 0` never would be evaluated, since the value is not needed, and hence the computation will terminate. *Lazy evaluation* is an implementation technique for implementing non-strict semantics. It is therefore slightly inaccurate to call any language with non-strict semantics lazy. However, the term lazy is so suggestive that it has caught on.

Laziness has two ingredients:

(i) An argument to a function should be evaluated only when its value is needed, not when the function is applied.

(ii) An argument should only be evaluated once; further uses of the argument within the function should use the value computed the first time. This scheme will give the same result as re-evaluating the argument because of referential transparency.

The first part is precisely what is achieved by normal-order reduction. Implementation of the second part is discussed in the next section.

Outermost (leftmost) reduction order is called *applicative-order* reduction or *eager* evaluation, and leads to *call-by-value* or *strict* semantics. This is what is found in most imperative languages (e.g. Ada, C, Modula 2, Pascal, and Smalltalk) and also in some functional languages (e.g. ML and Hope). In such languages the conditional function above, for example, cannot be properly implemented by the user (at least not by using the basic function call mechanism). Instead it must be supplied by the language implementor in some form.

Since normal-order reduction terminates whenever applicative-order does, but not vice versa, normal-order reduction is sometimes said to have greater expressive power. In a sense, a lazy language can also be said to be more declarative than a strict one since the user to a large extent can forget about the evaluation order because of the nice termination properties of normal-order reduction.

For another small example of laziness, consider

```
ones = 1 : ones;

main = take 3 ones;
```

where `take` is a function that takes the n first elements from a list. Here, `ones` is conceptually an *infinite* list of ones, but because of laziness, only as much of it as is needed will be evaluated, which in this case is the first three elements. The result of the program is thus `[1, 1, 1]`. Under a strict evaluation scheme, the program would have entered an infinite loop when trying to evaluate the list `ones`.

2.3.4 Implementation Strategies for the Lambda Calculus

There appear to be two main ways to build an implementation of the lambda calculus. The first is an environment-based scheme in which the parameters to a lambda abstraction are stored in an environment where they can be accessed when needed to calculate the value of the body. This is basically the same method as used in conventional imperative languages where parameters to functions and procedures are stored in a stack frame for later retrieval. This approach leads to the SECD machine [Lan64]. An efficient implementation of ML, built around the Functional Abstract Machine (FAM), which can be considered as an optimized SECD machine, has been constructed [Car84].

However, the environment based scheme seems to be best suited for languages with strict semantics [FH88, p. 241]. Also, it is probably not so suitable for parallel implementation of lazy functional languages [PJ87, pp. 379, 409–430].

The second approach to lambda calculus implementation, *graph reduction*, appears to be better suited both for implementation of languages with non-strict semantics and for parallel implementations. The G-machine is based on graph reduction, and since the G-machine was chosen as a basis for the Freja implementation, we will not consider environment based implementations further.

2.3.5 Graph Reduction

Graph reduction is a much newer technique than the environment based scheme and was first invented by Wadsworth [Wad71]. The basic idea is to represent lambda expressions as graphs, since graphs are much more convenient for computers to manipulate than text, and then evaluate the expressions in much the same way as the lambda expression in section 2.3.2 was evaluated.

Consider the lambda expression `(λx.+ 1 x) 4` again. This may be represented in graph form as follows:

```
       @
      / \
     \x  4
     |
     @
    / \
   @   x
  / \
 +   1
```

Here, `@` represents an *application node*, which is the graph equivalent of juxtaposition, and \ denotes λ. Normal-order reduction selects the top-most application node for reduction, since this is the outer-most, left-most redex in this case. The application node selected for reduction is known as the *root* of the redex.

Performing the reduction yields

```
      @
     / \
    @   4
   / \
  +   1
```

where `4` has been substituted for free occurrences of `x` in the body of the lambda abstraction.

Recall that we want to implement lazy evaluation, and that we therefore must avoid evaluating an expression more than once. This is achieved by physically overwriting the root of the redex with (the root of) the result. Thus, if the root occurs as a subexpression in several places, all occurrences will simultaneously be replaced by the value of that subexpression, and the subexpression will therefore never have to be evaluated again.

There are two problems associated with overwriting the root of the redex. Firstly, the result may be physically larger than the root. Secondly, the result may itself be shared (if it is not newly constructed), in which case overwriting the root would entail making a copy of the (root of the) result, which in turn could lead to loss of laziness if the result could be further evaluated. These problems can be overcome by overwriting the redex root with an indirection node that points to the result instead [PJ87, pp. 213–218].

Returning to the example, we find that both arguments for the addition are available, so the graph can be immediately reduced to `5`. Again, the redex root (which is the top application node) should be overwritten by (an indirection to) the result. If both arguments had not been available, we would simply have kept the graph unchanged as described in section 2.3.2. Hence

```
    @
   / \
  +   1
```

represents a partial application, which in this case is a function that adds one to its only argument. See Peyton Jones [PJ87, pp. 207–219] or Field & Harrison [FH88, pp. 241–272] for a more in depth treatment of graph reduction.

2.3.6 Implementing Recursion

In section 2.3.1 it was claimed that the basic lambda calculus is sufficiently powerful to express any computable function. This may seem strange, since there is, at first sight, nothing corresponding to recursion in the basic lambda calculus. The problem is that lambda abstractions are anonymous functions and hence cannot refer to themselves by name, which is the usual mechanism by which recursion is implemented.

However, it is well known that it actually is possible to express recursion in the pure lambda calculus by means of the so called `Y`-combinator, which is defined as:

```
Y = (λh.(λx.h (x x)) (λx.h (x x)))
```

This is useful because `Y` can be used to solve *fixpoint equations* of the type `x = f x` in that `x = Y f` is a solution to the equation, i.e. `Y f = f (Y f)`.

This property of the `Y`-combinator can be used to define recursive lambda abstractions by taking out the function that should be called recursively as an extra parameter, and then applying `Y` to the resulting abstraction.

To see this, suppose that we want to define the factorial function and that `F` is a lambda abstraction implementing this function, given that the free variable `fac` is bound to the factorial function (i.e. to the very abstraction we are trying to define). Thus, to form the desired abstraction, take out `fac` as an extra argument and apply `Y` to the result, giving:

```
Y (λfac.F)
```

which is an abstraction implementing the factorial function.

This is a very interesting theoretical result, but for practical purposes it is more efficient to give names to lambda abstractions. This effectively lets the functional programmer add new constants (i.e. named lambda abstractions) to the lambda calculus.

2.3.7 Template Instantiation and Compilation

When we described how to evaluate an application of a lambda abstraction in section 2.3.5 above, we said that the parameter should be substituted for free occurrences of the bound variable in the body. This statement has to be interpreted with some care; literally doing this would be a breach of referential transparency, since the lambda abstraction might be shared, and if it ever was applied to another argument, an erroneous result would be produced.

Therefore, a copy of the body should to be made, substituting the argument for free occurrences of the bound variable in the process. That is, the original lambda expression serves as a template which is instantiated with different values for the parameter.

To perform an instantiation, a tree walk has to be performed over the body of the lambda abstraction at runtime. Now, while it is perfectly possible to build a graph reduction interpreter based on template instantiation, the instantiation routine would in practice be a serious performance bottleneck, at least unless assisted by specialized hardware.

It is possible to do considerably better by noting that, for a given lambda abstraction, the same steps are performed every time the instantiation routine is invoked, the only difference being the value of the argument. Therefore, a fixed code sequence, that will perform the instantiation when executed, can be compiled in advance, provided it can access the argument somewhere at runtime. This is the basic idea behind compilation of functional languages.

Unfortunately, it is not possible to compile a fixed code sequence for all lambda abstractions. For example, consider the following:

$$(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{+\ x\ y}))$$

Here, `x` is free in the inner lambda abstraction. This means that whenever the whole expression is applied to a single argument, the result will be a *new* lambda abstraction, for which obviously no instantiation code could have been compiled in advance in the general case. For example:

$$(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{+\ x\ y}))\ \mathtt{1} \Rightarrow (\lambda\mathtt{y}.\mathtt{+\ 1\ y})$$

$$(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{+\ x\ y}))\ \mathtt{42} \Rightarrow (\lambda\mathtt{y}.\mathtt{+\ 42\ y})$$

This can be solved in two ways. The first approach is to associate an environment with the lambda abstractions where values of free variables can be found at runtime. Hence a fixed code sequence can be compiled since the code is parameterized with respect to its free variables. (Note that accessing free variables is different from accessing parameters, cf. access of global variables in e.g. Pascal.) This leads back to the environment-based scheme that was described in section 2.3.4.

The second approach is to eliminate free variables by transformation, which can be done at the expense of performing simultaneous application of lambda abstractions to multiple arguments. This leads to *supercombinator reduction*, which is the topic of the next section.

2.3.8 Supercombinators and Lambda-lifting

Note that the problem above never would have arisen if the lambda abstraction had been applied to two arguments and these simultaneously had been substituted into the inner body, since the lambda abstraction as a whole has no free variables. Let us, therefore, first define those lambda abstractions that are amenable to such multiple-argument application:

> A *supercombinator* `S` of arity n ($n \geq 0$) is a lambda expression of the form
>
> $$\lambda\mathtt{x}_1.\lambda\mathtt{x}_2.\cdots\lambda\mathtt{x}_n.\mathtt{E},$$
>
> where `E` is not a lambda abstraction, such that
>
> (i) `S` has no free variables,
>
> (ii) any lambda abstraction in `E` is a supercombinator.

If all parameters to a supercombinator are substituted into the supercombinator body simultaneously, and if a supercombinator is only applied once all its parameters are available, it is possible to compile a fixed code sequence that will create an instance of the supercombinator body when executed, since the supercombinator as a whole does not have any free variables.

Fortunately, any lambda abstraction can be transformed into a supercombinator by first abstracting out any free variables as extra parameters, and then replacing the occurrence of the lambda abstraction with the supercombinator applied to the free variables. This transformation is known as lambda-lifting, and it is discussed in more detail in Peyton Jones [PJ87, pp. 220–231]. Thus supercombinators can be used as a practical implementation basis.

It is convenient to give (arbitrary) names to the supercombinators that are created by the lambda-lifting algorithm. This also permits direct implementation of recursion as discussed in section 2.3.6.

We will illustrate the basic method by means of a small example. Consider:

```
(λx.+ x ((λy.* y x) 2))
```

The inner lambda abstraction is not a supercombinator since `x` occurs free. Therefore `x` should be abstracted out as an extra parameter. Giving the arbitrary name `FOO` to the resulting supercombinator, and replacing the occurrence of the inner lambda abstraction with `FOO` applied to the free variable `x` yields:

```
FOO = (λx.(λy.* y x))
(λx.+ x (FOO x 2))
```

The original lambda abstraction is now in supercombinator form as well, and by assigning the arbitrary name `FIE` to it we get:

```
FOO = (λx.(λy.* y x))
FIE = (λx.+ x (FOO x 2))
```

To emphasize the special status of supercombinators, they are often written in a lambda free form:

```
FOO x y = * x y
FIE x = + x (FOO x 2)
```

In this form, the supercombinators can be understood as rewrite rules. Executing the code that builds an instance of the supercombinator body, can therefore be interpreted as performing a rewrite of a supercombinator application.

To summarize, by performing lambda-lifting, a user program will be turned into a set of supercombinators. These can then be compiled, which will yield efficient instantiation of lambda abstractions. In a sense, the compiler constructs a specialized machine-language coded combinator interpreter from each program. This is the basis of the G-machine, as described in the next section.

2.3.9 The G-machine

The G-machine [Aug84, Aug87, Joh84, Joh87] is an abstract, finite-state machine. It is designed to be a suitable target for supercombinator compilation, yet its architecture is close enough to that of an ordinary computer to make implementation of the G-machine simple and efficient on standard von Neumann hardware.

The state of the G-machine is a 4-tuple <S, G, C, D>, where:

(i) S is the spine stack (see below),

(ii) G is the graph,

(iii) C is the code sequence that remains to be executed,

(iv) D is the dump, a stack on which S and C are stored to preserve them across 'subroutine' calls.

The G-machine and its instruction set, the G-code, can be formally defined by specifying the state transition for each of its instructions.

As explained in section 2.3.7, executing a compiled code sequence should build an instance of the corresponding supercombinator body. Suppose we have the following supercombinator:

```
SUB5 x = - x 5
```

When applied to an argument, say `SUB5 4`, which corresponds to the graph

```
    @ $
   / \
 SUB5 4
```

the following graph should be built:

```
    @ #
   / \
  @   5
 / \
-   4
```

Furthermore, the root of the redex (indicated by `$`) should be overwritten by the root of the result (indicated by `#`). When a G-code sequence is invoked, (reference(s)) to the argument(s) can be found on the top of the spine stack and (a reference to) the root of the redex below. In our case (with the stack top to the left) the spine stack will look like:

{`4`, `<root of the redex>`}

PUSHINT 5	Push the integer 5	{5, 4, <r>}
PUSH 1	Push copy of argument	{4, 5, 4, <r>}
PUSHGLOBAL -	Push subtract function	{-, 4, 5, 4, <r>}
MKAP	Build application node	{@ - 4, 5, 4, <r>}
MKAP	Build application node	{@ (@ - 4) 5, 4, <r>}
UPDATE 2	Overwrite redex root	{4, @ (@ - 4) 5}
POP 1	Tidy stack, remove arg.	{@ (@ - 4) 5}
UNWIND	Continue execution	

Figure 2.2: G-code for building graph for (- x 5).

The G-code sequence in figure 2.2 will construct the desired graph. The numbers after PUSH and UPDATE are offsets from the stack top. The number after POP are the number of elements to remove. In practice, as was indicated above, the stack only contains references to the built structures. The structures themselves are part of the graph.

We will conclude this section by briefly discussing how a reduction is initiated. When the EVAL instruction is executed, the G-machine inspects the top element on the stack. If this is a data object (e.g. an integer) then evaluation is complete. If it is an application node, the current spine stack and code sequence are saved on the dump and the evaluation process initiated. The G-machine will now follow the left branch of the application node, pushing further application nodes onto the stack as they are found. This is known as unwinding the spine (hence spine stack), since a chain of application nodes look a bit like a spine (the application nodes themselves are called the vertebrae and the arguments are known as the ribs):

```
      @
     / \
    @
   / \
  @
 / \
```

When a supercombinator is encountered, the G-machine will check if enough arguments are available. If not, evaluation is complete and the old spine stack and code sequence are restored from the dump. Otherwise, the G-code sequence corresponding the supercombinator is executed next. Once completed, the unwinding process is reinitiated (by means of the UNWIND instruction), and this continues until no more evaluations can take place (too few arguments available or the whole thing has evaluated to a data object). Then, the old spine stack and code sequence are restored from the dump and execution continues from where it left off.

Further details on the G-machine may be found in the G-machine papers or in Peyton Jones [PJ87] or Field & Harrison [FH88].

2.4 Algorithmic Debugging

The algorithmic debugging technique was first developed in the context of Prolog [Sha82]. In previous research by our group, the algorithmic debugging method has been generalised to a class of imperative languages and its bug finding properties improved by integrating the method with program slicing [Sha91, FGKS92, KSF92, Kam93]. In the following, the basic principles of algorithmic debugging are described. To simplify the presentation, we will assume strict evaluation order in this section.

Algorithmic debugging is based on the notion of an *externally visible bug symptom*, i.e. an execution of a program did not produce the expected result. This must have a cause: either the topmost procedure did something wrong, or some procedure invoked from the topmost one produced some erroneous result. By inspecting the invocations performed from the topmost procedure, it is possible to determine what is the case: if all invocations yielded correct results, then the topmost procedure must be at fault. Otherwise, one of the invocations exhibits some visible bug symptom, and we may thus apply the above reasoning over again, tracing the source of the bug down through the call hierarchy. Since the debugger cannot know what is a bug symptom and what is correct behaviour, at least not initially, there must be an *oracle*, i.e. the user, ready to supply it with this information.

Given a visible bug symptom, algorithmic debugging guarantees that the procedure or function containing the bug will eventually be found, provided that the user answers the questions about the program behaviour correctly. If there is more than one bug in a program, only one of them will be found. But once the first bug is removed, algorithmic debugging may be applied again in order to find the next bug.

In more detail, algorithmic debugging proceeds as follows. The debugger first executes the program and builds an *execution tree* at the procedure level. A node in the tree is constructed for each procedure invocation, and essential trace information, such as the procedure name and the values of all input and output parameters, are recorded. If any further procedure calls are made during the current invocation, these become children of the current node. New children are inserted from left to right as the corresponding procedures are invoked.

Once the execution has finished, the algorithmic debugger starts searching for the bug by traversing the execution tree in a preorder manner. For each node, the debugger interacts with the user by asking whether the behaviour of the procedure invocation corresponding to the node is correct. The user may reply yes or no to this. If a positive answer is given, the search continues from the next branch to the right of the current one, otherwise the search continues from the leftmost child of the current node (if it has any). The search ends at a node exhibiting incorrect behaviour when one of the following two conditions holds:

(i) No further procedure calls were made during the procedure invocation corresponding to this node; i.e. this node has no children.

```
insert x []     = [x];
insert x (y:ys) = y:(insert x ys), if y > x
                = x:y:ys,          if x < y
                = y:ys,            otherwise;

sort []     = [];
sort (x:xs) = insert x (sort xs);

main = sort [2, 1, 3];
```

Figure 2.3: Erroneous insertion sort program.

(ii) All procedure calls performed during the procedure invocation corresponding to this node fulfil the user's expectations.

Finally, the debugger reports the name of the offending procedure.

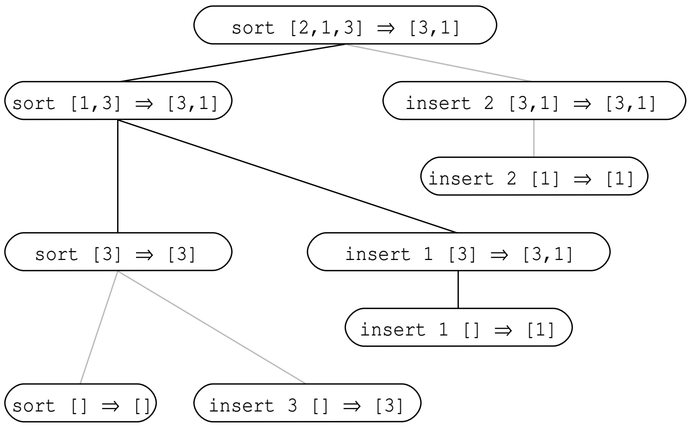
The user's answers are remembered in a database (which could be preserved across debugging sessions) so that the same question does not have to be asked twice.

Figure 2.3 shows an erroneous insertion sort program. The bug is in the function `insert`, the arguments to `>` are the wrong way round.

In figure 2.4 we can see how this program may be debugged algorithmically. First, the program is run and the execution tree is constructed. Then debugging starts, at the root of the tree. Since the user answers no to the first two questions, the debugger moves down to the leftmost child of `sort [1,3]`. According to the user, the behaviour recorded in this node is correct, so the branch to the right is investigated next. The behaviour of that node is wrong, but its only child behaved correctly. The bug must therefore be in the body of the procedure that is associated with this node, i.e. in `insert`. Paths in the tree that were not explored are shown in grey.

Note that, to be able to give a correct answer, the user must be given the 'full picture' for each question, i.e. the computation must depend only on its input parameters, and there must be no effects of the computation besides what is indicated by the output parameters. Procedures that fulfil this criterion are said to be *side-effect-free*. It is possible to achieve side-effect-freeness in programs written in imperative languages by program transformations [Sha91].

Note also that programs that abort due to runtime errors, as well as programs that loop, easily can be handled within the algorithmic debugging framework (in the latter case provided that the user interrupts the program) provided procedures and functions conceptually are allowed to return bottom, $\perp$, representing a diverging computation.




```
sort [2,1,3] ⇒ [3,1]?
>no
sort [1,3] ⇒ [3,1]?
>no
sort [3] ⇒ [3]?
>yes
insert 1 [3] ⇒ [3,1]?
>no
insert 1 [] ⇒ [1]?
>yes
Bug located in function "insert".
```

Figure 2.4: Execution tree and debugging session for the insertion sort program.

Chapter 3

Lazy Debugging

This chapter gives an introduction to debugging for lazy functional languages. First, in section 3.1, the basic problems of debugging in a lazy context are explained in some detail, motivating why specialized debugging tools are needed for this class of languages. Then, in section 3.2, a number of approaches that have been suggested or tried earlier are outlined. Lastly, in section 3.3, based on the observations made in the two preceding sections, we will try to justify the basics of our own approach.

3.1 The Need for Lazy Debugging Tools

Consider the following functional program, where `foo` is a function that clearly must not be applied to the empty list:

```
foo xs = 0,     if hd xs < 0
       = hd xs, otherwise;

fie xs = (foo xs) : fie (tl xs);

main = fie [-1];
```

The problem is that there is no termination condition in the recursive function `fie`, which means that `fie` eventually will apply `foo` to the empty list since `main` applies `fie` to the finite list `[-1]`. Suppose that we have strict semantics. Then we would get an execution tree as shown in figure 3.1.

In the rightmost branch, we see how the application of `hd` to the empty list provoked a runtime error, represented by the symbol $\perp$ (' bottom'). At this point, we simply have to follow the edges from the leaf node towards the root, which in practice is easily achieved by inspecting the runtime *call* stack, in order to find out that the problem is that `fie` has applied `foo` to `[]`, the empty list.

Now suppose that we had had lazy semantics instead. We would then get an execution tree as shown in figure 3.2. Applying the same technique

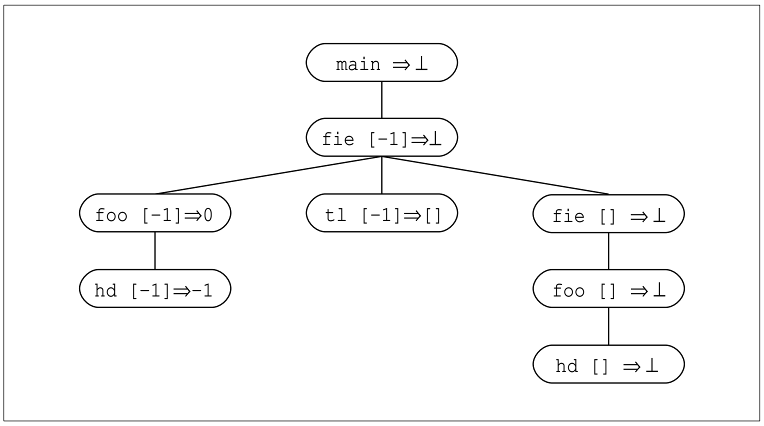


Figure 3.1: Strict execution tree.

as above to this tree would not give any insight as to what the problem might be: **foo** is applied to an expression that will evaluate to the empty list, that is for sure, but which function applied **foo** to that expression? There is no recollection of this in any of the execution tree nodes on the path from the node that caused the error to the root node, which are the only nodes available to a conventional debugger in the form of the runtime stack. The key difference is that we now have a *demand* stack, rather than a call stack, due to the demand driven execution model.

The presence of partially evaluated expressions in the lazy execution tree should also be noted. In general, these may become very large and complicated, even when they only denote very simple values, thus making it even harder for a user to get a clear understanding of what is going on.

As a case in point, in anticipation of what is to come, consider trying to apply algorithmic debugging to the lazy tree. Clearly, it is a lot harder to answer whether

```
fie (tl [-1]) ⇒ foo (tl [-1]) : fie (tl (tl [-1]))
```

is correct or not, than answering the corresponding strict question

```
fie [] ⇒ ⊥
```

We could also try to do a conventional trace of the lazy execution:

```
Entering fie:xs = [-1]

Leaving fie, result is: (foo [-1]) : (fie (tl [-1]))

Entering foo:xs = [-1]

...
```

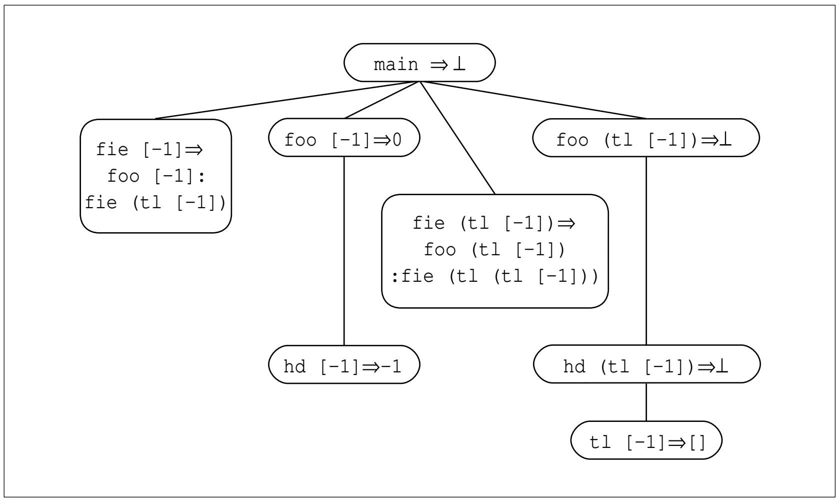


Figure 3.2: Lazy execution tree.

Apparently, `fie` did not call `foo`! This behaviour is probably not in agreement with the programmer's mental model of the execution process, and it is at least not obvious from reading the source code. In any case, lazy evaluation permits the programmer to largely forget about evaluation order, so there is a strong case for not bothering him with it during debugging.

To sum up, debugging of lazy functional programs is made difficult by delayed evaluation, counter-intuitive evaluation order and partially evaluated expressions. Because of this, traditional debugging tools and techniques are of little or no use. Hence 'lazy' debugging tools, that address the above mentioned problems and that conceptually fit into a lazy framework, are needed, even though these might still be based on conventional ideas (e.g. tracing).

3.2 Related Work

In this section, some proposed debugging approaches for lazy languages are reviewed. This is by no means neither an exhaustive survey, nor a very deep and detailed one, but it probably covers the most important ideas in the field.

3.2.1 Hall & O'Donnell

Hall & O'Donnell [HO85, OH88] propose a number of approaches to debugging for lazy functional languages in two articles. Their focus is on implementing debugging tools within an interactive, purely functional environment, the main argument for this being portability. They use the language Daisy, a lazy descendant of Lisp.

One suggested approach is to transform the source code of the entire program so that it produces a trace of its execution as well as its normal value. The trace should then be printed before the value of the program so that it is still possible to see something, even if the program happens to loop.

The main problem with this, as Hall & O'Donnell also point out, is of course that the very printing of the trace might turn an otherwise terminating program into a non-terminating one, e.g. if the trace contains references to infinite data structures or to diverging computations, the values of which would not usually be needed. Furthermore, their method presupposes that all types of value, including functional ones, can be printed in a sensible way from within the language. This is not necessarily the case in all functional languages (e.g. Miranda), thus the portability argument is somewhat undermined.

Hall & O'Donnell therefore suggest another approach where the user by means of an interactive debugging function may traverse and evaluate the target program, change variable bindings, evaluate and print expressions in various contexts, etc. Thus the user is allowed to observe, control and modify the execution of a program, all within a purely functional environment. To achieve this, access is needed to the system `eval` function. Also, if the user happens to invoke a non-terminating computation, e.g. by asking for some infinite structure to be printed, it must be possible to interrupt the computation and recover.

Clearly, however, there is still a problem with printing. If the target program is written in a lazy style, chances are that infinite structures and diverging computations will occur frequently, thus making debugging a rather elaborate process involving a lot of trial, error and interrupt. One could conceive special support within the language, e.g. a non-evaluating printing mechanism, but such an *ad hoc* extension would at the best have very dubious semantics (for a start, referential transparency would be lost).

3.2.2 Toyn & Runciman

Toyn & Runciman [TR86, Toy87] argue that even if it is nice to have interactive programming environments, the need for debugging tools for compiling language implementations cannot be neglected. They propose a system in the context of a combinator reduction machine, where the graph is annotated in such a way, and the reduction mechanism modified accordingly, that the computation history of the various pieces of graph on the heap is always maintained as annotations in the graph. Thus it is possible, at any time instant, to take a 'snapshot' of the computation, i.e. to present a finite source-level textual representation of the state of the computation. A snapshot would typically be taken when the program aborts due to a runtime error, or when the user interrupts it because it has entered an infinite loop.

It is less obvious how one should get a good, revealing snapshot on a computation that terminates but produces the wrong result. In his thesis, Toyn [Toy87] proposes traditional bottom-up testing to deal with this case.

3.2.3 Kishon *et al.*

Kishon *et al.* [KHC91, Kis92] have taken a formal and systematic approach to debugging as well as other monitoring activities, e.g. profiling. Starting from a denotational continuation semantics of an arbitrary language, they automatically derive a monitoring semantics by composing the language specification with a monitor specification. For example, by composing with a tracing specification, the meaning of a program in the language would be changed to be a trace in addition to its original value.

The key difference between this approach and Hall's & O'Donnell's (in the context of lazy functional languages), apart from the former being more systematic and general, is that the computation in this framework is observed from outside the language, which means that there is no problem with printing of infinite structures, etc. On the other hand, unevaluated values will show up in the trace instead, necessitating some kind of work around to get rid of them. Sturrock [Stu92] has tried to use Kishon's framework to build a strictifying algorithmic debugger for a lazy functional language but reports problems.

3.2.4 Kamin

Kamin [Kam90] starts from an operational semantics of a lazy language and changes it so that a program in the language has a tree-structured trace of its execution as its meaning. However, in contrast to Kishon's system, Kamin relies on a meta-evaluation rule to get rid of as many unevaluated values as possible. The rule simply states that values should be shared, i.e. they should be represented by pointers to unique heap-allocated objects. Thus, when the computation has terminated, any value will be seen in its most evaluated form. Further, the execution tree is constructed so that its structure reflects the syntactic structure of the program being debugged, as would be the case in a strict language. Together with the meta-rule, this effectively abstracts away most of the operational details of lazy evaluation.

The general goal of Kamin's work is to demonstrate a hypertextual approach to trace-based debugging. The emphasis is therefore placed on providing the user with facilities for browsing through the trace, visualizing large data structures etc.

3.2.5 Hazan & Morgan

Hazan & Morgan [HM93] take a source-level transformational approach to debugging. A program is transformed by their tool so that an explicit call path is constructed and passed around during execution. The path records the static call structure, i.e. it contains information that would be present on a call stack in an implementation of a strict language. However, it contains only the names of the functions, not any parameter values.

The idea is then that the path is appended to any error messages in the code, so that, in the event of a program error, in addition to the error message, one can see which instance of a function invocation that caused the problem. Note that if the path also contained parameter values, there would again be problems with printing if there happened to be any infinite structures, etc. Though clearly somewhat limited in scope, the tool has been found useful and it is used extensively in the maintenance and development of a large Miranda system (about 12 000 lines of code).

3.2.6 Henhapl *et al.*

Another debugger that has been used in practice is part of a programming environment for rapid prototyping developed at Technische Hochschule Darmstadt by Henhapl *et al.* [HKS91]. This debugger, however, is conventional in the sense that it is based on setting breakpoints and single stepping, and it does not address the problem of unevaluated function arguments and results.

3.2.7 Sparud & Johnsson

Our work is made in loose co-operation with Sparud & Johnsson from the functional programming group at Chalmers University of Technology[1] where there is also an ongoing effort to construct a debugger, in their case for Haskell [Spa94]. Their approach differs from ours in several interesting ways, though there are also many similarities.

Their approach is also based on tracing by source-level transformation. First, a tool (written in Haskell) transforms the program that should be debugged so that a trace in the form of an execution tree is constructed in addition to its normal result. The program is then run, producing a trace which the user is allowed to browse through by means of a graphic user interface based on the functional Fudget library [CH93]. This means that the debugger is purely functional, which is very interesting.

The structure of the tree is determined by the syntactic structure of the program, so the Chalmers approach resembles Kamin's work in this respect. Also, because they are working within the language, values will be seen in their most evaluated form since inspection forces evaluation of any left unevaluated parts. The effect is, up to a point, the same as that achieved by Kamin's meta-rule, the difference being that forcing evaluation of a value that was not evaluated during the normal course of execution always risks non-termination, as was noted above concerning the work of Hall & O'Donnell. Another difference between Hall & O'Donnell and the work at Chalmers is that the latter, through a more sophisticated transformation scheme, gets round some of the problems related to printing functional values from within the language.

A particular problem of doing tracing by source-level transformation in a language such as Haskell is its rigid Hindley-Milner type system: unless

[1] Within the same NUTEK funded project.

one is prepared to make all functions have the same return type, it is rather difficult to type the trace. The same problem would occur in languages such as Miranda and Freja. However, by employing so called *existential types*, which, while not being part of standard Haskell, are present in the Chalmers Haskell compiler, it is possible to solve the problem.

3.2.8 Naish

Finally, the algorithmic debugger by Naish [Nai92] should also be mentioned. This debugger is for a lazy functional language which is implemented by translation into Prolog by means of the flattening transformation, which converts an evaluable function with n arguments into a predicate with $n+1$ arguments. For debugging purposes, the transformation is changed so that the resulting clauses, in addition to the functional result, also produces an execution tree with all necessary trace information.

To deal with unevaluated expressions, Naish suggests that they should be replaced by universally or existentially quantified variables, thus significantly reducing the size of the questions posed to the user. An unevaluated expression that occurs in an argument position should be replaced by a universally quantified variable, whereas an unevaluated expression that occurs only in the result should be replaced by an existentially quantified variable.

To illustrate this, consider the function `take` that returns the first n elements of a list, and the function `from` that returns the infinite list consisting of all integers from some specific integer and upwards:

```
take 0       xs       = [];
take (n + 1) []       = [];
take (n + 1) (x : xs) = x : take n xs;

from n = n : from (n + 1);
```

Now, suppose that the following computations, among others, took place:

`take 2 [3,4:(from (4 + 1))]` ⇒ `[3,4]`

`from 3` ⇒ `[3:(from (3 + 1))]`

where the notation `[`$\mathtt{x}_1$`, `$\mathtt{x}_2$`, ..., `$\mathtt{x}_n$` : exp]` suggests a partially evaluated list whose as yet unevaluated tail is given by the expression `exp`.

It would be possible to debug algorithmically by asking a user about the above computations, but as we saw in section 3.1, such a question could become very unwieldy. In Naish's approach, the questions would be simplified and posed as follows:

For all `A`, is it true that `take 2 [3,4:A]` ⇒ `[3,4]`?

For some `A`, is it true that `from 3` ⇒ `[3:A]`?

3.3 Some Observations

First, it should be noted that most of the approaches mentioned in the previous section are based on tracing in some form. Kamin even argues that tracing might well be inevitable in the context of lazy functional languages.

However, a raw trace reflecting operational details will not be terribly useful. This is because one of the ideas behind lazy languages is to let the user forget about such details, and because what happens probably is too far removed from what is conceptually happening or what the user thinks is happening if he were to explain it. This is in contrast with imperative languages in which the user has to understand what is going on in operational terms, which means that a debugger based on this is not infeasible.

Thus, a more abstract view is needed, as provided by e.g. Kamin and Sparud in their respective work. This is exactly why strictification was introduced in our debugger, and it is interesting to note that the execution tree that results from doing strictification on a lazy tree will be very similar to what is achieved by Kamin or Sparud, even though the method the current LADT implementation is using to do strictification is very different from what they are doing. (The improved tracing scheme outlined in chapter 5 is closer though.)

Based on this and our own experience with the early version of LADT that did not do strictification, we believe that the construction of a strictified execution tree is instrumental to successful debugging for lazy functional languages, be it algorithmic or by other means.

Naish's approach might be an alternative to strictification, but we think that strictification probably will yield questions that are simpler and relate more closely to the way the programmer thinks.

Second, there are two ways to go about tracing: either from within the language by means of source-to-source transformations or from the outside, by modifying the language implementation in some sense. Both ways have their merits.

The main advantage of doing tracing by source-to-source transformation is perhaps portability. A debugger based on this is not tied to any particular implementation of a language (provided it does not make use of any non-standard extensions of the language implementation in question, of course), and it may even be easily adaptable to other languages. Furthermore, being able to do debugging entirely within a lazy functional framework is conceptually rather appealing.

The price that has to be paid for this is loss of control: there is no way to tell an evaluated value from an unevaluated one when working within the language, so there is always the risk of starting some non-terminating computation merely by inspecting the trace, and in many cases it will not be possible to recover from such a situation. Should the problem actually be that the program loops, it might not even be possible to start debugging. Also, a piecemeal tracing scheme, as mentioned in the introduction, by which only a part of the trace is stored at any one time, other parts being created on demand by re-execution, is probably very difficult to achieve by program transformations.

Thus, for practical reasons, we think it is better to change the language implementation so that it produces a trace as a side-effect. When working at the language implementation level, it is very easy to distinguish evaluated values from non-evaluated ones and communicating this to the debugger. Thus there is no risk of inadvertently causing non-termination. Moreover, it is straightforward to deal with non-termination at this level, e.g. by catching `CTRL-C` and start debugging on the parts of the trace that have been built up to the point where the program was interrupted. Finally, we think this is the way to go to implement piecemeal tracing. The price in this case is a tracer that is tightly bound to a particular language implementation.

We thus believe that the implementation level tracing approach taken in LADT, as described in the following chapter, is justified.

Chapter 4

LADT

This chapter describes our debugger, the Lazy Algorithmic Debugging Tool (LADT). In section 4.1 the principles behind it are outlined. A small debugging example is then presented in section 4.2. In section 4.3, the implementation of the debugger is considered in some detail. The chapter concludes in section 4.4 with a small evaluation of the system, where its major shortcomings are pointed out. This serves as a motivation for the refined implementation ideas in the following chapter.

4.1 Lazy Algorithmic Debugging

In this section, we describe how algorithmic debugging may be applied to programs written in lazy functional languages.

4.1.1 The Basic Approach

Basic algorithmic debugging, as outlined in section 2.4, may readily be used for lazy functional languages since functions in such languages *are* side-effect-free. We do have to regard the execution environment of a function as belonging to its input parameters, though, since functions may reference free variables.

It should also be noted that not even interactive programs need any special treatment, since I/O is performed purely functionally. Typically, the input is made available as a list of characters and supplied as an argument to the top function, whose result will be the output from the program. Thus, an interactive, lazy functional program is no different from any other lazy functional program as far as the debugger is concerned, which is very convenient of course.

Contrast this with languages where I/O is performed by side-effects. Data read from or written to an external device, would then have to be made visible as extra procedure arguments and results by means of program transformations or corresponding techniques.

A first algorithmic prototype debugger could therefore be constructed rather quickly. However, while gaining experience in using this early version of the debugger, the fact that arguments to functions in general are partially evaluated expressions soon proved to be a major problem: the user is usually concerned with the values that these expressions represent, whereas details of the inner workings of the underlying evaluation machinery often are of little or no interest. Furthermore, the questions that the debugger asked were frequently textually very large and difficult to interpret.

On the one hand, this problem might have been especially apparent in our case, since our compiler does not do strictness analysis, which is something any good compiler for a lazy language would do. On the other hand, strictness analysis is only an approximation, and there will always be occasions when unevaluated expressions are passed as parameters but used later (otherwise we would not need the lazy semantics in the first place).

All this suggest that we should replace unevaluated expressions by the values they represent, wherever possible, to give the user an impression of strict evaluation, which probably is closer to the user's mental model of the evaluation process anyway. This is what we call *strictification*.

However, strictification is, as illustrated in the following subsection, not as straightforward as it first might appear: not only must values be substituted for expressions wherever possible, but the actual structure of the execution tree has to be changed as well. Otherwise, there is no longer any guarantee that the bug will be found.

4.1.2 Why Substitution of Values for Expressions Is Not Enough

Suppose that strictification was implemented in the obvious way, i.e. before asking whether a function application yielded a correct result or not, any unevaluated expressions that occur in the arguments or in the result of the application are replaced by the results of evaluating the expressions, in case they are known to the debugger (i.e. were needed at some point during the execution). This, of course, has to be done recursively, should the results themselves contain any unevaluated subexpressions. The user will then see a version of the function application which is as strict as possible, given a particular execution trace.

Unfortunately, the debugging algorithm is then no longer guaranteed to find the bug, as illustrated by the example below (the function `add` is incorrect):

```
dbl x = add x x;

add x y = x * y;

main = dbl 3;
```

If no strictification is performed, evaluating main would yield an execution tree as depicted in figure 4.1. The debugger quickly concludes that the bug

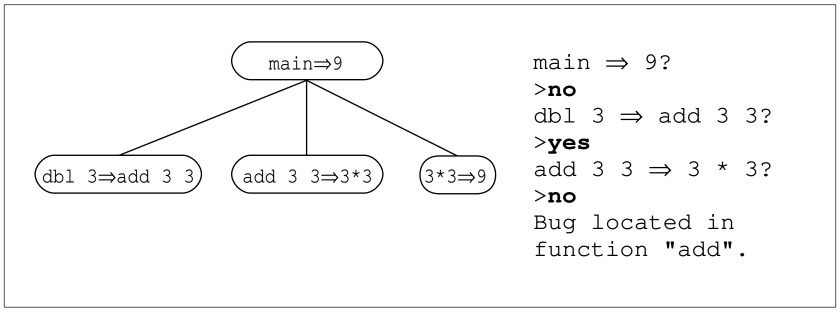


Figure 4.1: Lazy execution tree.

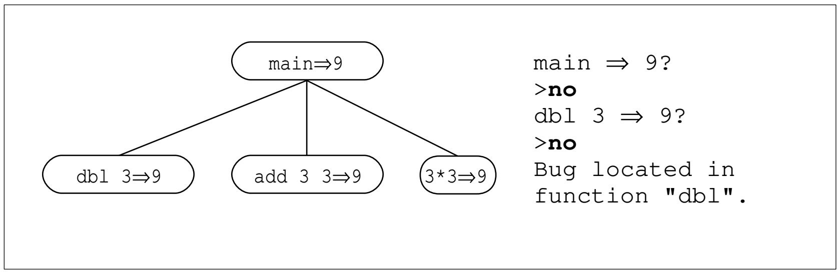


Figure 4.2: Incorrectly strictified execution tree.

must be in the function `add` since applying `add` to `3` and `3` yields something erroneous and since that node has no children.

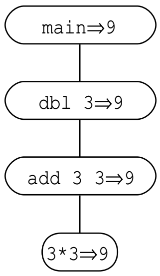
Now, suppose that we did substitute values for expressions wherever possible in the tree above, and then tried to do algorithmic debugging. The result is shown in figure 4.2. When asked whether `main` should evaluate to `9`, the user answers no, whereupon the debugger proceeds to the first child node and asks whether `dbl 3` should evaluate to `9` or not. Since the intention is that `dbl` should double its argument the user again answers no. Now, since this node has no children, the debugger will come to the conclusion that the bug is within the function `dbl`, which is wrong.

The problem is that by doing the substitutions, in this case first substituting `3*3` for `add 3 3` and then `9` for `3*3`, we are effectively pretending that these computations take place at an earlier point in time than is actually the case, but this is not reflected in structure of the execution tree.

A correctly strictified tree and the resulting debugging interaction may be seen in figure 4.3. Note how the nodes involved in the substitution have become child and grand child of the `dbl 3` node.

4.1.3 Correct Strictification

A correct way of doing strictification can be derived from the following two observations:




```
main ⇒ 9?
>no
dbl 3 ⇒ 9?
>no
add 3 3 ⇒ 9?
>no
3 * 3 ⇒ 9?
>yes
Bug located in
function "add".
```

Figure 4.3: Correctly strictified execution tree.

- Performing a substitution of the result of evaluating an expression for the expression in one of the arguments of a function application, corresponds to evaluation of the expression in question *before* entering the function (precisely what happens in a strict language). Thus, a new node should be inserted in the execution tree to the left of the node corresponding to the function application.

- Performing a substitution of a result for an expression in the result of a function application, corresponds to evaluation of the expression in question *during* the invocation of the function (again as in a strict language). Thus, the node corresponding to the function application should be given a new child; inserting it to the right of its siblings will do.

These transformations should be applied recursively, i.e. the new nodes must themselves be strictified. The process is depicted in figure 4.4 for a simple case. Grey nodes represent computations that have taken place elsewhere; `E1`, `E2` and `E3` are expressions.

However, there is no need to actually perform these transformations on the execution tree; it is sufficient to ask the questions during the debugging phase in such an order as if the transformation had been performed. This is how strictification is presently implemented in LADT.

4.1.4 Algorithmic Debugging with Strictification

The algorithm for algorithmic debugging with strictification is given in pseudo code in figure 4.5. The implemented debugger, in addition to yes and no answers, also supports ‘maybe’ answers and a kind of simple assertions, details of which are omitted from the pseudo code for reasons of simplicity. These features are instead outlined informally.

Also missing from the pseudo code are checks for preventing attempts to strictify nodes in the execution tree that already are being strictified. This can be achieved by keeping nodes currently under strictification in a list. Then, before strictification of a new node is started, it is checked whether it

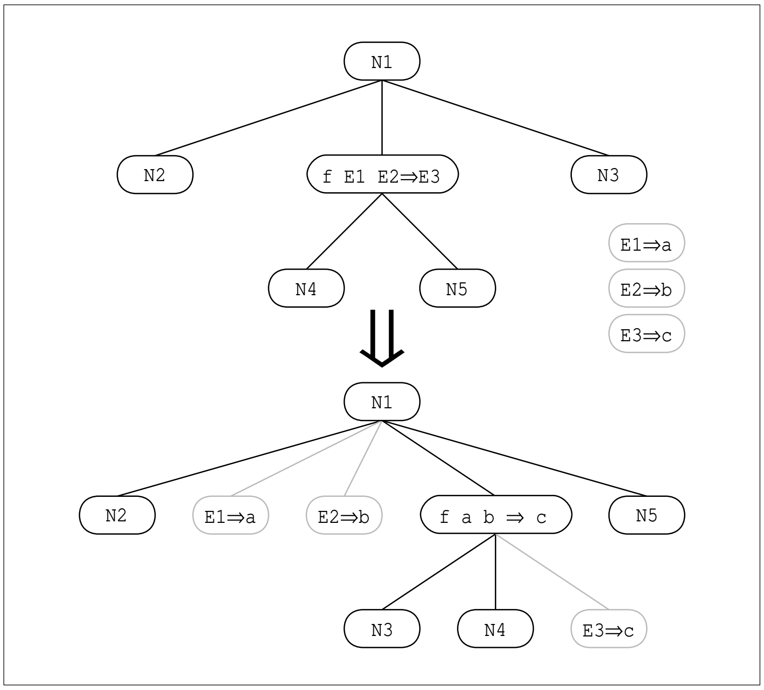


Figure 4.4: Example of correct strictification

is already present in the list or not. If it is, it can safely be ignored. Similar arrangements must be provided for the evaluation procedures below.

Two abstract data types are assumed: **exec_tree** and **expr**. An execution tree node consists of a function application and its result (both of type **expr**) and zero, one or more children. These are accessed by means of the functions **App**, **Res** and **Children**. A function **Tree** that returns the execution tree corresponding to a function application is also supposed to exist. This function can, of course, only be used if the application was evaluated during the execution, which is checked by the function **Has_Been_Evaluated**.

An expression is either an application of a function to some arguments; a constructed data object (e.g. a tuple) having zero, one or more fields; or some atomic object such as an integer. The functions **Args** and **Fields** are used to access the arguments of an application and the fields of a constructed object, respectively.

There is also a database that can be queried about the status of a function application. Initially, the status is 'unknown'. This is changed to 'correct' whenever the user answers that an application behaved correctly. Thus the same question need not be asked twice. The remaining function and procedure names should hopefully be self-explanatory.

There are, of course, cases when the user might be unwilling to supply a definite yes or no answer to a question, either because he really does not

```
PROCEDURE Strictify_And_Debug(tree: exec_tree)
VAR strictified_app, ev_res: expr;
BEGIN
   IF Status_Is_Unknown(App(tree)) THEN
      strictified_app := Make_App(Fun_Name(App(tree)),
                             MAP Evaluate_And_Debug Args(App(tree)));
      ev_res := Evaluate(Res(tree));
      Pretty_Print(strictified_app); PRINT "=>";
      Pretty_Print(ev_res); PRINT "? ";
      IF User_Answer = "YES" THEN
         Set_Status_To_Correct(App(tree));
      ELSE
         FOR ALL c IN Children(tree) DO
            Strictify_And_Debug(c);
         ENDFOR;
         Evaluate_And_Debug(Res(tree));
         (* If we get here, no bug in children. *)
         PRINT "Bug located in function ";
         PRINT Fun_Name(App(Tree)); PRINT ".\n";
         EXIT; (* Exit immediately when bug found. *)
      ENDIF;
   ENDIF;
   (* If we get here, the behaviour was correct *)
END;

FUNCTION Evaluate_And_Debug(e: expr): expr
BEGIN
   IF Is_App(e) THEN        (* Function application *)
      IF (Has_Been_Evaluated(e)) THEN
         Strictify_And_Debug(Tree(e));
         RETURN Evaluate(Res(Tree(e)));
      ELSE
         RETURN Make_App(Fun_Name(e), MAP Evaluate_And_Debug Args(e));
      ENDIF;
   ELSEIF Is_Constr(e) THEN (* Tuple, CONS-cell... *)
      RETURN Make_Constr(Constr_Name(e), MAP Evaluate_And_Debug Fields(e));
   ELSE                     (* Integer, character...*)
      RETURN e;             (* Already fully evaluated. *)
   ENDIF;
END;

FUNCTION Evaluate(e: expr): expr
BEGIN
   IF Is_App(e) THEN        (* Function application *)
      IF (Has_Been_Evaluated(e)) THEN
         RETURN Evaluate(Res(Tree(e)));
      ELSE
         RETURN Make_App(Fun_Name(e),MAP Evaluate Args(e));
      ENDIF;
   ELSEIF Is_Constr(e) THEN (* Tuple, CONS-cell... *)
      RETURN Make_Constr(Constr_Name(e), MAP Evaluate Fields(e));
   ELSE                     (* Integer, character...*)
      RETURN e;             (* Already fully evaluated. *)
   ENDIF;
END;
```

Figure 4.5: Algorithmic debugging with strictification.

know the answer, or because the question seems to be large and complicated. Thus, there is an option to answer '*maybe* this is correct, I don't know'. This effectively lets the user postpone answering the question and get on with the debugging in the hope that a bug is positively identified before he is faced with this question again.

Initially, a 'maybe' answer is treated as 'no'. Thus the children of the current node will be searched. If no bug is found, the algorithm will eventually get back to the node that the user was unsure about, knowing that all its children behaved correctly. At this point, the user is again asked about the node. If she this time indicates that the result is correct, debugging will proceed as usual, the only difference being that the user has had to answer more questions than otherwise would have been the case. If the answer is that the result is incorrect, then there must be a bug in the applied function. However, if the users insists on answering 'maybe', the algorithm is forced to conclude that there might be a bug in the function applied. This is reported to the user and the algorithm then continues as if the result was correct, since it is still possible that a more definite bug will be found.

The assertion facility gives a user the possibility to suppress questions regarding a particular function (i.e. asserting its correctness). This has proved to be a quite convenient feature, but obviously it is very primitive as far as assertions go.

4.2 Algorithmic Debugging of a Small Program

The Freja program in figure 4.6, that is supposed to calculate the first five prime numbers by using the Sieve of Erathostene, contains a bug in the function `not_div_x` defined locally in `sieve`; the operator equal (`==`) should be replaced with the operator not equal (`~=`):

The interaction between the user and the debugger, when the above program is executed, is shown in figure 4.7. The questions have been numbered for easy reference. The user may answer `y` for 'yes', `n` for 'no' and `m` for 'maybe' to the questions. She may also issue commands that control various printing options and give assertions; `h` for 'help' gives a list of all available options. Note that missing arguments in applications (i.e. instances of partial applications) are indicated with an underscore (`_`) and that an ellipsis (`...`) is used to represent tails of long lists and tails that are not evaluated. Also, note that variables belonging to the environment of a function are shown in a `where`-clause (see for example question 8). The notation `foo.fie` indicates that function `fie` is declared locally in function `foo`.

Clearly, the numbers 2, 4, 8, 16 and 32 are not the first five prime numbers, so the answer to question 1 has to be no. The same holds for questions 2 and 3 as well. The application in question 4 reduces to a list of numbers starting with 2 and 3, which seems to be correct. However, `sieve` applied to this list should evaluate to a list of prime numbers, which

```
shownums []     = [];
shownums (x:xs) = shownum x ++ " " ++ shownums xs;

filter p []       = [];
filter p (x : xs) = x : filter p xs, if p x
                  = filter p xs, otherwise;

from n = n : from (n + 1);

take 0       xs       = [];
take (n + 1) []       = [];
take (n + 1) (x : xs) = x : take n xs;

sieve (x : xs) = x : sieve (filter not_div_x xs)
                     where
                        not_div_x y = (y mod x == 0);
                     ;

primes = sieve (from 2);

main = shownums (take 5 primes);
```

Figure 4.6: Erroneous Sieve of Erathostene.

is obviously not the case in question 5. The answer to this question is therefore again no.

To be able to answer question 6, the user first requests unevaluated list tails to be shown (the command `uet`). Since `(from 3)` reduces to a list starting with 3 and having `(from 4)` as its tail, this reduction is correct.

The expression in question 7 should filter out everything divisible by 2 from the list of natural numbers starting from 3. Apparently, however, we are left with all even numbers from 4 and upwards, so this is wrong. Finally, in question 8, the offending function is found. Since 3 is not divisible by 2 the expression ought to yield `true`.

For comparison, an unstrictified version of question 2 is given in figure 4.8. It is not too hard to infer the answer to this question, but it is clearly not as straightforward as above. The result of the reduction is a list whose head is the first prime number (2) and whose tail is an expression that according to its specification *should* evaluate to the remaining prime numbers. Therefore, the answer to the unstrictified version of the question is yes, whereas the answer to the strictified question is no since the tail *in fact* evaluates to something erroneous:

A larger example of lazy, algorithmic debugging may be found in appendix B.

```
2 4 8 16 32

1) Did the program behave correctly?
(y/n/m/h)? n

2) main => "2 4 8 16 3..."
(y/n/m/h)? n

3) primes => [2,4,8,16,32,...]
(y/n/m/h)? n

4) (from 2) => [2,3,...]
(y/n/m/h)? y

5) (sieve [2,3,...]) => [2,4,8,16,32,...]
(y/n/m/h)? n

6) (from 3) => [3,...]
(y/n/m/h)? uet
(from 3) => [3:(from ($plus 3 1))]
(y/n/m/h)? y

7) (filter (sieve.not_div_x _ where x=2)
           [3:(from ($plus 3 1))]) =>
[4,6,8,10,12,14,16,18,20,22,...]
(y/n/m/h)? n

8) (sieve.not_div_x 3 where x=2) => False
(y/n/m/h)? n

Bug located in function sieve.not_div_x
```

Figure 4.7: Debugging the faulty Sieve of Erathostene.

```
main =>
['2':
 ($append
    [' ']
    (shownums
        (take 4 (sieve (filter
                          (sieve.not_div_x _ where x=2)
                          (from ($plus 2 1))))))))]
(y/n/m/h)? y
```

Figure 4.8: Unstrictified version of question 2.

4.3 Implementation of LADT

In this section the implementation of our debugger is discussed. The modifications of the compiler needed to support algorithmic debugging are outlined as well as the implementation of the debugger itself. A technical problem that makes it very hard to achieve perfect strictification in the current implementation is also discussed.

4.3.1 Modifications of the Compiler

As noted in section 4.1.1, free variables must somehow be dealt with. Being a G-machine implementation, Freja is based on graph reduction using supercombinators. A source program containing functions with free variables is transformed into supercombinator form by lambda-lifting, as was explained in section 2.3.8. Since only basic lambda-lifting is performed (rather than *fully lazy lambda lifting*, see e.g. Peyton Jones [PJ87, pp. 245–259]), followed by η-reduction and removal of redundant supercombinators, there is a one-to-one mapping between the functions in the source program and the supercombinators in the object program. This is very convenient, but in a more realistic language this might not be the case, and one would also have to take other source constructs (e.g. list comprehensions) into account.

Therefore, by implementing the algorithmic debugging on the supercombinator form of a program, no extra work is needed to take out free variables; this is already done as a part of the basic compilation process. To support algorithmic debugging, the compiler only has to add some extra debugging information to each supercombinator definition and insert calls to tracing routines in the object code so that an execution tree may be built at runtime. Note that this has to be done for system supplied supercombinators as well, even though we know they are correct, since we need a complete trace tree to perform strictification during the debugging phase.

The extra information that is needed consists of the name of the function corresponding to the supercombinator, the arity of the supercombinator, the number of free variables that have been taken out as extra parameters and the names of those variables. The information is needed so that questions from the algorithmic debugger may be phrased in terms of the original source program rather than in terms of the transformed program, which would be very inconvenient for the user.

Calls to two different tracing routines are inserted in the code of a supercombinator, one call at the beginning and one just before each possible exit point. The routine that is called first creates a node in the execution tree and records the values of all parameters (including the abstracted free variables); the second routine records the result of the supercombinator application. Thus one node in the execution tree will be created whenever a supercombinator reduction is initiated, which happens exactly when the corresponding function is applied to enough arguments, i.e. at least as many arguments as the arity of the function.

Obviously, a debugger must be able to cope with a program that terminates in an abnormal manner or fails to terminate. In case of a program error (which terminates execution immediately), a call is made to a special routine that first fixes any tree nodes for which no results as yet have been recorded and then immediately invokes the algorithmic debugger. Nodes are fixed by inserting a special result, 'error', which is semantically equivalent to a diverging computation, i.e. bottom, $\perp$. If a program goes into an infinite loop, the user may force termination by pressing `Ctrl-C`. The tree is then fixed in the same manner as if a program error had occurred.

The Freja compiler consists of some 10 000 lines of C-code (not counting blank lines or comments). In this respect, the modifications needed for supporting algorithmic debugging were fairly minor, a rough estimate is that about 400 lines had to be modified or added.

4.3.2 The Algorithmic Debugger

LADT itself is implemented in about 3 000 lines of C. The debugger is linked with the compiled Freja program that is going to be debugged, forming a single executable. When it is run, the Freja program is first executed and the execution tree built. Then the debugger is invoked and algorithmic debugging begins.

The debugger consists of two main parts: routines for building the trace tree, and routines for performing the actual debugging. The tree is constructed by making calls to the two trace routines, as described above. Arguments to supercombinators and returned results are all pieces of graph located on the functional program's own private heap. To preserve them for the execution tree (recall that redexes are physically overwritten in the course of graph reduction), we have chosen to copy them off the heap, though other schemes could be devised. The copying algorithm must be able to handle circular graphs; otherwise, the debugger might end up in an infinite loop.

To facilitate equality testing on graphs, equal graphs are mapped into the same storage using a hashing algorithm, i.e. before any graph node is built, it is checked to see whether an equal graph node already exists. Thus pointer comparison can be used to test for graph equality within the debugger, and the pointers themselves can be used for further hashing when indexing the execution tree in order to perform the strictification reasonably efficiently.

4.3.3 Why Perfect Strictification Is Not Achieved

Since the debugger has recorded the results of all function applications that have been evaluated during the program execution, it should theoretically be possible to do a perfect strictification transformation, i.e. a transformation that yields a tree in which all values are in their most evaluated from. Unfortunately, this is quite difficult achieve in practice when strictification is performed the way it is currently done.

Consider an unevaluated application `foo (1+2)` (that would evaluate to

7, say). Suppose further that the value of `foo (1+2)` is indeed needed later on, but that the subexpression `(1+2)` is shared with some other expression, the value of which is needed before the value of `foo (1+2)` is needed. Then the debugger would record that the result of `foo 3` is `7`, but it would know nothing about the result of `foo (1+2)`. Therefore, it is not possible to substitute `7` for this application during strictification, unless the debugger is able to figure out that `foo 3` and `foo (1+2)` really means the same thing. However, achieving this in the general case without forcing evaluation (and risking non-termination) would be a very expensive search problem indeed.

In an attempt to somewhat compensate for this flaw, the debugger explicitly evaluates some system function applications in a few special cases when this is safe (typically arithmetic functions such as `+` when all arguments are available and evaluated). Clearly, better strictification would be preferable.

4.4 Evaluation of LADT

4.4.1 Experience in Using the Debugger

LADT has so far been used to successfully debug a number of toy programs, including programs calculating Fibonacci numbers and solving the eight queens problem, as well as a somewhat larger program to evaluate arithmetic expressions (around 250 lines of code).

The number of questions that on average has to be answered in order to find a bug of course varies drastically with the size and type of the program. Finding a bug in the expression evaluator seemed to require some 50–60 questions to be answered on average. There is clearly room for further improvements here.

4.4.2 Problems with the Current Implementation

The present LADT implementation has two severe problems that make it difficult or impossible to apply to real, large programs. The main one is the size of the trace tree: currently every single reduction is recorded, which means that the size easily could become hundreds of megabytes. The situation is akin to running a lazy functional program without garbage collection. Also, building the tree takes a long time. Though we have not done any extensive measurements, execution seemed to be slowed down by two or three orders of magnitude when tracing. This means that it might take a long time before debugging even can start.

Now, it would certainly be possible to keep the trace on secondary storage, even if tracing (as well as strictification) then would become even more expensive. One might even argue that the sizes of primary memories within a not too distant future will be large enough to accommodate traces of such sizes. However, the fundamental problem is that there is *no upper bound* on the size of trace, and that problem remains however we store it.

The other big problem is that far too many questions are asked. This makes LADT a very tedious debugging tool to use. This is especially so since it is often obvious to a user that a large number of the questions are irrelevant: it might well be the case that a user can see exactly what is wrong in a result returned from function. If this information could be conveyed to the debugger, rather than just a simple yes or no answer, it should be possible to get to the point quicker, thus avoiding many irrelevant questions.

Yet another problem with the current implementation of the debugger is that perfect strictification is not achieved. This means that the user sometimes will have to deal with unevaluated expressions when there should not have been any need to do so. While static strictness analysis probably would make the problem less apparent, strictification is still better than just using static strictness analysis for our purposes.

Chapter 5

Ideas for Practical Implementation

Despite its problems, we have found the debugger useful, and it should even be possible to debug quite large programs using it, as long as the test cases are of reasonable size. Furthermore, we think that the algorithmic approach to debugging is basically sound and promising in the context of lazy functional languages. But we would obviously like to do better. Thus, in this section, a number of ideas that address the problems of the current implementation and extend the basic approach are outlined, with the aim of making algorithmic debugging for lazy functional languages practical.

We start, in section 5.1, by stating a number of requirements that we believe a realistic debugger must meet, in greater or less degree. The various ideas for achieving this are then outlined in the sections that follow.

5.1 Requirements

5.1.1 Debugging Real Programs

As has been explained earlier, the current LADT system suffers from several shortcomings, which means that we do not consider it to be a realistic debugging tool for 'real-world' programs. Partly based on our experience with LADT, we believe the following to be a reasonable set of basic requirements for a practically useful algorithmic debugger:

- It must be possible to debug any program that does not exhaust the memory resources during normal execution, regardless of the size of the full trace. (*Some* working space must obviously be set aside for the debugger, however.)

- The debugger must handle abnormally terminating as well as non-terminating programs.

- The number of irrelevant questions asked during debugging should be kept to a minimum.
- Debugging must take place at the source-level, even if programs are subject to complex transformations during the compilation process, which often is the case when high level source constructions, e.g. list comprehensions, are employed.
- A good user interface is important. It should provide easy-to-use browsers permitting the exploration of source code and large data structures, and it should not burden the user by showing unnecessary trace details.

While LADT meets some of the above requirements, it is still a long way from being a realistic debugger.

An interesting case is that of source-level debugging. LADT currently meets this requirement, but that is mostly because Freja is a very simple language with few advanced constructs, and because of the simplistic Freja implementation. Together, these two aspects mean that there is a one-to-one correspondence between functions in the source program and generated supercombinators. In the presence of for example list comprehensions or anonymous functions (lambda abstractions), things would get considerably worse. More sophisticated compilation might have the same effect.

In sections 5.2 through 5.7 a number of ideas for meeting the basic requirements above are outlined. Of these, the most fundamental ones are:

- *Thin tracing*: if it is known beforehand that some functions or modules are correct (e.g. library routines), it should not be necessary to trace everything, thereby reducing the size of the trace as well as the number of questions asked.
- *Piecemeal tracing*: do not store the entire trace at once. Instead, parts of the trace can be constructed as and when needed by rerunning the program that is being debugged.
- *Immediate strictification*: instead of building a lazy execution tree and applying strictification on it afterwards, a strictified tree could be built directly.
- *Slicing*: if the user is able to be more specific as to what is wrong, rather than just saying that *something* is wrong, it ought to be possible to make use of this information to reduce the number of questions asked by applying program slicing.

While these ideas, in their current form, still cannot *guarantee* that any program can be debugged with some predetermined space overhead (thus violating the first requirement), they do allow any program to be debugged as long as there is room to store the largest execution tree node encountered during debugging, and then still room run the program to completion. Given a reasonably sized heap, we do not expect any problems in this respect in practice. On the other hand, frequently having to rerun a program might turn out to be a new practical problem.

5.1.2 Cognitive Requirements

Up until now, nothing much has been said about alternatives to algorithmic debugging: it has more or less been assumed that algorithmic debugging is a good way to navigate through the execution tree. However, it is not at all clear that an average user is prepared to submit herself to such a rigid and inflexible search strategy: the *cognitive plausibility* of the debugging approach must not be ignored.

In her thesis, Ducassé [Duc92, pp. 43–50] reviews a number of cognitive studies regarding debugging, and, based on these, formulates some cognitive requirements. Basically, she concludes that debuggers must be flexible and adaptable, that they must abstract away irrelevant or uninteresting execution details, and that they must help the user to understand the program that is being debugged, e.g. by providing source-code browsers etc.

These are quite reasonable requirements, but while it would be straightforward to meet some of them (e.g. providing a source-code browser), it is less clear how one should go about meeting some of the others, especially in our case where we have the additional problem of laziness to deal with. We will discuss alternative debugging strategies and cognitive aspects a bit further in section 5.8.

5.2 Thin Tracing

5.2.1 Avoiding Unnecessary Tracing

Presently, LADT records every single reduction that takes place during execution. Quite a few of these are applications of language primitives (e.g. `+`, `if`) which should be correctly implemented. Moreover, large systems are built modularly, so it will often be the case that there are a large number of well tested, fairly trusted modules and a few, new, 'prime suspects' when a bug manifests itself. The very least, library modules may reasonably be trusted. Thus, it should be possible to reduce the size of the trace considerably, since there is no need to ask a question for which it is known that the answer will be 'yes, this is correct', which in turn obviates the need to record information for the question.

So under a *thin tracing* scheme, only a subset of the reductions would be traced, based on assumptions regarding the correctness of certain modules and language primitives. Clearly, this will also reduce the number of questions that are asked and the time taken to build the tree. However, not having the entire trace at our disposal, means that strictification cannot be performed as it is currently done, i.e. during the debugging phase. Therefore, a new approach to strictification must be adopted to use thin tracing.

5.2.2 Correctness and Higher Order Functions

There is also a more subtle problem as to what is meant by a function being 'correct' in this context. For a first-order function it is obvious: a function is correct if it computes the expected result for whatever arguments it is applied to. During debugging, this means that a user no doubt would indicate that an application of the function is correct, and the entire branch in the execution tree emerging from that point may thus be cut away.

But for a higher-order function that takes another function as an argument, it would be a rather bold claim that the result is correct for arbitrary arguments: the supplied function is effectively behaviour that has been abstracted out of the higher-order function, and claiming that the higher-order function produces the correct result when some arbitrary behaviour is being plugged back in cannot be justified. It is only possible to say that the higher-order function uses the supplied function in the intended way.

For our purposes, this means that the branch in the execution tree corresponding to the application cannot be cut away completely: any applications of the supplied function must be verified.

On the other hand, if it is known that the function that is supplied as an argument is correct as well, then the application could be treated as in the first-order case. This suggests that some simple 'correctness calculation' should be performed on higher-order applications. However, we have not as yet looked into these issues in any depth.

5.3 Piecemeal Tracing

5.3.1 Trading Space for Time

Even if it is possible to substantially reduce the size of the trace using thin tracing, there is still no guaranteed upper bound on the size of the execution tree. Indeed, as long as the trace for a whole execution is going to be stored, there can be no general, fixed such upper bound.

An interesting alternative would then be to store only so much trace as there is room for. Debugging is then started on this first piece of trace. If this is enough to find the bug, all is well. Otherwise, the program to be debugged is re-executed, and the next piece of the trace is captured and stored. Indeed, for our purposes, the program could be regarded as just a compact coding of the execution tree, thus allowing us to trade space for time by decoding it repeatedly. We will refer to such a tracing scheme as *piecemeal tracing* from now on.

Re-executing the program is not a problem, since pure functional programs are deterministic, but any input to the program must obviously be preserved and reused. Also, if the user aborts the execution since she suspects that the program loops, the debugger must automatically abort the program at the same point when it is later re-executed.

The process is illustrated in figure 5.1. The large, dashed triangle corresponds to the entire execution tree, whereas the small, topmost triangle

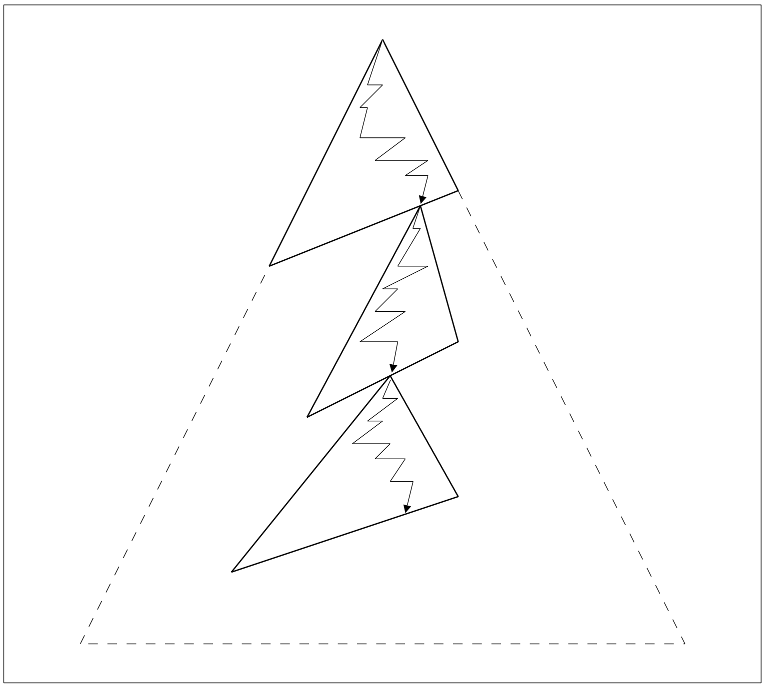


Figure 5.1: Piecemeal tracing.

corresponds to the part of the trace that is built during the first execution. Debugging begins at the top of the triangle and proceeds down along the indicated path. Once the fringe of the stored trace is reached, assuming no bug has been found, the program is re-executed and the trace corresponding to the second triangle stored. Debugging then resumes and proceeds downwards until the fringe is reached again, whereupon the program is re-executed once more, and so on, until the bug has been found.

Due to the high cost of trace construction (see section 4.4.2), a piecemeal scheme might actually be beneficial from a time perspective as well, since the time overhead of tracing should be significantly reduced if only a small portion of the trace is constructed. The user would therefore be able to start debugging quicker. Furthermore, if the overhead is large enough, and the program is only rerun a few times, the scheme might turn out to be cheaper overall. Thus, there is a fair chance that the space-for-time trade-off really will turn out not to be a trade-off at all.

5.3.2 The Query Distance

The question is, then, how to select the piece of the trace to be stored in a sensible way? Just storing reductions as they are performed until the trace storage is full would not be very useful, since it may then happen that a very deep but narrow piece of the tree is captured. If the top reduction is

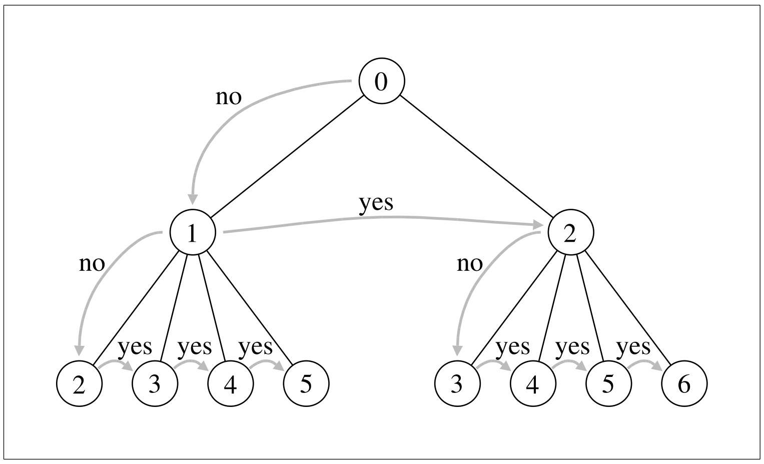


Figure 5.2: The query distance.

actually correct, the program would have to be rerun immediately to get the next piece of the trace. It would be better if trace corresponding to the next n questions, regardless of what the answers to these questions are, could be stored.

This leads to the idea of introducing a distance measure on the nodes in the tree, based on the number of questions that would have to be answered to get from one node to another (if this is possible, otherwise the measure is undefined). We term this measure the *query distance*, and the idea is illustrated in figure 5.2. The nodes are labelled with their query distance from the root node, and the grey arcs are labelled with the answer that would take a user from one node in the execution tree to the other during algorithmic debugging.

5.3.3 Capturing the Right Part of the Tree

Given a cache for execution tree nodes, piecemeal tracing can be implemented as follows. Suppose that a particular node in the execution tree has been reached during debugging, and that it is found that more trace is needed to proceed. Call this particular node the *current node*.

Now the entire program is re-executed[1]. Only nodes reachable from the current node (i.e. nodes for which the query distance is defined relative to the current node) are stored in the cache. When the cache is full, nodes with the largest query distance relative to the current node are thrown out in favour of nodes with a smaller query distance. Nodes corresponding to the next n questions will thus be in the cache when the tracing is completed, where n depends upon the size of the cache.

[1] There is a good reason for re-executing the *entire* program, as will become clear in section 5.4.3.

Thus, an arbitrary upper bound may be imposed on the number of tree nodes stored at any one time. Obviously, the larger the size of the cache, the fewer times the program will have to be rerun: a trade-off that may be adjusted to suit any particular requirements. However, as is the case with thin tracing, the entire trace is no longer available during debugging, which means that strictification cannot be performed as it is currently done. This problem is addressed in the next section.

5.4 Immediate Strictification

5.4.1 Avoiding Unnecessary Work

During tracing, our current debugger physically copies pieces of the graph from the heap to construct the trace. The copied graph pieces thus become dead: any further rewriting of the original graph pieces will not be reflected in the trace. This, and the non-strict structure of the execution tree as such, is what necessitates the post-execution strictification pass.

This means that the debugger in effect, by substituting values for expressions, is redoing work that has already been done during the execution of the program that is being debugged. Furthermore, as discussed in section 4.3.3, some strictification opportunities are missed by LADT, which means that only approximate strictification is achieved. Also, doing strictification afterwards is only possible if the entire trace, containing every single reduction, is available to the debugger. Thus thin and piecemeal tracing cannot be integrated with the current implementation of the debugger. Finally, copying is expensive.

Why not, then, avoid the copying by simply keeping pointers to the graph pieces, and thereby also gaining substitution of values for expressions for free, eliminating the need for a post-execution strictification pass? Recall that redexes are physically overwritten once reduced, which means that values will be seen in their most evaluated form once the execution is completed.

While we indeed would get substitution for free by such a scheme, achieving strictification is not simply a matter of maintaining pointers, since the tree structure would remain unchanged resulting in incorrect strictification (see section 4.1.2). But the idea is very attractive, and by constructing the tree directly in a way that mirrors the syntactic structure of the source program, it is possible to make use of it. Not only will it then not be necessary to have access to the complete trace during debugging, since post-execution strictification is not needed, but it will also remove the problem of approximate strictification. Further, it opens up possibilities for piecemeal and thin tracing. We call this scheme *immediate strictification*.

5.4.2 Getting the Tree Structure Right

The basis of our method is the observation that at the instant when a lazy language builds a redex for (potential) later evaluation, a strict language would evaluate that expression directly. Thus, in the strict case, a new child of the node representing the current function invocation would be created in the tree at this moment, provided we were tracing the execution. To construct a tree with a strict structure in the lazy case, the allocated redex should therefore be annotated with a reference to the node in the execution tree representing the current function invocation, so that the node created if and when the redex is reduced can be inserted at the correct place in the tree.

Conceptually this is very similar to what is done in other trace based approaches that involve live values in their traces, e.g. the work by Kamin [Kam90] or that by Sparud & Johnsson [Spa94]. However, in our case the tracing takes place at the language implementation level, which gives us fine control over the process and allows us to do it piecemeal, something we think is very important.

Maintaining references from tree nodes to live pieces of graph on the heap is easy: just keep a pointer to the graph (and make sure that the garbage collector knows about it). The references that go the other way round, from redexes to tree nodes, are more problematic. This is because tree nodes may be thrown away due to the piecemeal tracing process, as described below, invalidating a large number of references on the heap.

Our current solution to this problem is to assign a unique number to each reduction that takes place, thereby also to each node in the complete execution tree, and refer to the nodes in the tree by this number via a hash table. This means that it is no longer necessary to explicitly invalidate references on the heap to tree nodes when what they are referring to is thrown away: it can always be discovered later that a reference has become invalid. Note that once it is discovered that a parent is missing, tracing should be disabled for that part of the tree.

5.4.3 Integrating Piecemeal and Thin Tracing

Piecemeal tracing is achieved by pre-allocating space for a fixed number of tree nodes. The nodes in the tree are annotated with a measure indicating their distance from the root of the tree, the idea being that nodes that are close to the root will be stored in preference to nodes that are more distant. Should it turn out that the heap becomes full due to the presence of graph pieces held only by the execution tree, the number of nodes in the tree may be dynamically adjusted down to a minimum of a single node.

The measure is more or less the query distance, as discussed in section 5.3.2, but for technical reasons, it might be that an estimate of it will have to make do. The problem is that the distance has to be calculated at redex construction time, i.e. before we know for sure whether the allocated redex eventually will be reduced and a corresponding tree node created.

This all means that the execution tree may be pruned during tracing. Wherever this takes place, a reference to the cut off branch in terms of the unique number of its root node is inserted. Then, if such a place is reached during debugging, the missing part of the tree can be recreated by rerunning the program, not starting tracing until the reduction with the correct number is about to take place.

Note that the easiest way to ensure that all values are seen in their most evaluated form, is to re-execute the entire program each time: if just the application that was cut off is repeated, the debugger would not know how far the evaluation was to be taken.

Implementing thin tracing in the above scheme (for first order functions anyway) is very easy: just don't do anything. In the code produced for traced functions, calls to the tracing routines have to be inserted so that tree nodes are created when an application involving any of the functions is reduced. For functions assumed or known to be correct, normal code can be generated: no calls to the tracing routines will be made and thus no nodes will be constructed for these functions, which is exactly what we want.

So far, the scheme outlined above has has been partially implemented in the form of a program demonstrating immediate strictification. The algorithms needed for implementing the full scheme, incorporating piecemeal and thin tracing, have however been developed in detail, and a test implementation is currently being done.

5.4.4 Example of Immediate Strictification

To make the above discussion a bit more concrete, we will try to illustrate immediate strictification by a small example. Suppose that we have a function `f`:

```
f x = x + x;
```

Further, suppose that we are calculating `f E`, where `E` is some as yet unevaluated expression returned as the result from an earlier application. The situation immediately after the reduction of `f E` is shown in figure 5.3. Note how copying is avoided by letting the argument and the result be live pieces of graph, referred to via pointers from the newly constructed execution tree node. Also note how the allocated redexes refer to their respective parent in the tree (the grey arrows).

The result of `f E` is a new expression, `E + E`, and the next thing that will happen is that this expression is reduced. Now, `+` is strict in both its arguments (i.e. an application of it cannot be evaluated unless the values of its arguments are evaluated first), which causes the evaluation of `E`, yielding `7`, say.

The situation is shown in figure 5.4. Note how the nodes corresponding to the reductions of the expressions `E` and `E + E` have been created below the tree nodes that originally created them, rather than below the node corresponding to the current function invocation. Also note how these expressions are physically overwritten by the result of evaluating them (see section 2.3.5), which means that all tree nodes referring to them will see them in their most evaluated form once execution has terminated.

Thus, under an immediate strictification scheme, execution tree growth will occur simultaneously at many places, which figure 5.5 tries to illustrate.

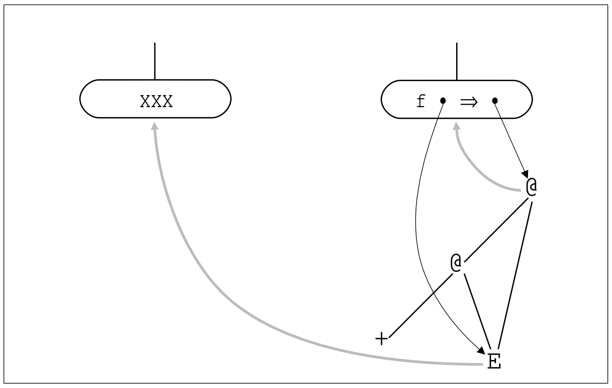


Figure 5.3: The situation after reduction of `f E`.

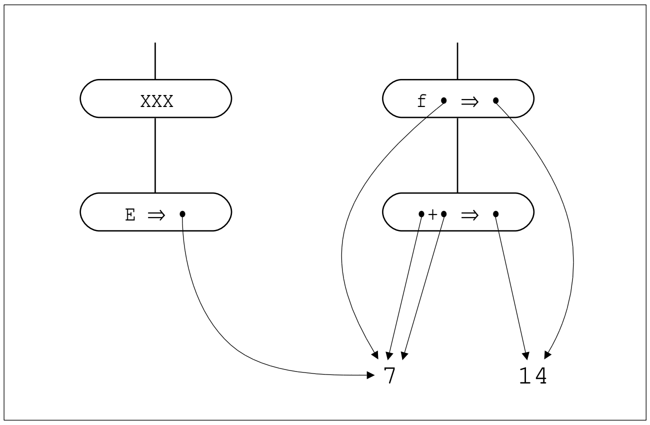


Figure 5.4: The situation after reduction of `E + E`.

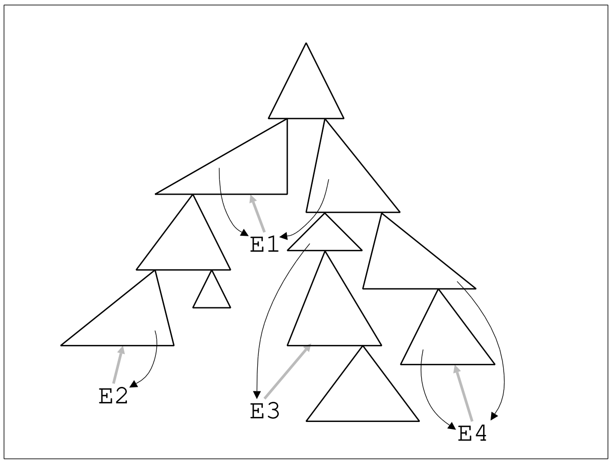


Figure 5.5: Tree growth under immediate strictification.

5.5 Source-level Debugging

Not being able to debug at the source-level, would clearly make the already difficult task of debugging even harder for the user. However, mapping execution time events to source constructs is not always that easy when a program has been subjected to sophisticated compile time transformations, e.g. for removal of syntactic sugar, or heavy optimization to improve performance, in both cases resulting in object code that is far removed from the original source program. Problematic constructs in this respect include list comprehensions.

Monad structured programs [Wad92] or programs employing lazy functional state threads [LPJ94] might also create problems, since textually large lambda abstractions, i.e anonymous functions, are involved. In the latter case there is also the added problem of the construct permitting truly mutable state at the language implementation level.

An obvious approach for dealing with the problems introduced by advanced compilation techniques, is to sidestep the whole issue by employing simpler transformations and turning off optimization when compiling for debugging purposes. This should be acceptable, as long as the simplistic compilation does not lead to such inefficiencies that the program run unacceptably slow or not at all.

In this section, as an example, it will be shown how simple list comprehensions may be debugged algorithmically by taking this approach. We think that other source level debugging problems could be solved in like manner.

5.5.1 List Comprehensions

List comprehensions, as any syntactic sugar, do not add any fundamental power to a functional language and are easily transformed into simpler language constructs as a part of the compilation process. However, they do increase the expressive power of the language considerably, and help in writing concise and clear programs. Hence list comprehensions are used extensively, which means that a debugger must handle them somehow.

In the following, we will deal with simple list comprehensions of the form

$$\texttt{[E | } \texttt{Q}_1\texttt{, } \texttt{Q}_2\texttt{, } \ldots\texttt{, } \texttt{Q}_n \texttt{]}$$

where `E` is an arbitrary expression and each *qualifier* $\texttt{Q}_i$ is either a *generator* of the form `v <- L`, where `v` is a variable, `L` is an expression yielding a list and `<-` means 'drawn from', or a *filter*, which is just a boolean predicate.

The idea is that elements are drawn one by one from the generator lists, bound to the corresponding generator variables, and then used to produce a new list element given by the expression `E`, unless filtered out by one of the boolean filter predicates. A variable introduced by a generator may be referred to in the expression `E` or in any qualifier to the right of the one that introduced the variable.

Some examples may make this clearer. The first yields a list consisting of the squares of all odd natural numbers from 1 and upwards:

```
[ x * x | x <- from 1, odd x ]
```

The following produces the 'cartesian product' of two lists, `xs` and `ys`:

```
[ (x, y) | x <- xs, y <- ys ]
```

For example, for `xs = [1, 2, 3]` and `ys = ['a', 'b']`, the result would be the list

```
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'),
 (3, 'b')]
```

Note that generators to the right changes more rapidly than those to the left.

For a final example, the following is a definition of a sorting function using the quicksort algorithm:

```
qsort []     = [];
qsort (x:xs) = qsort [y | y <- xs, y < x] ++
               [x] ++
               qsort [y | y <- xs, y >= x];
```

As the examples above show, list comprehensions may, in a sense, be regarded as the functional equivalent of loops, permitting iteration over lists.

5.5.2 Translation of List Comprehensions

We will now define the meaning of list comprehensions more formally, and show how they may be translated into nested function definitions. The presentation is closely based on Peyton Jones [PJ87, pp. 127–138].

We begin by defining reductions rules for list comprehensions. Comprehensions will be written in the form `[E | QS]` where `E` is an expression and `QS` is a sequence of zero or more qualifiers.

The following five rules suffice to define list comprehensions:

(i)	`[E	v <- [], QS]`	$\Rightarrow$	`[]`
(ii)	`[E	v <- E':L', QS]`	$\Rightarrow$	`[E	QS][v↦E'] ++` `[E	v <- L', QS]`
(iii)	`[E	False, QS]`	$\Rightarrow$	`[]`
(iv)	`[E	True, QS]`	$\Rightarrow$	`[E	QS]`
(v)	`[E	]`	$\Rightarrow$	`[E]`

The notation `[E | Q][v↦E']` in rule (ii) means `[E | QS]` with all free occurrences of `v` replaced by `E'`.

These reduction rules may be used to derive a translation of list comprehensions into normal function definitions. The first two rules suggest how to handle the case when the first qualifier of the comprehension is a generator.

Such a comprehension should be mapped into a function taking the list of the generator as its only argument. If the function is applied to the empty list, then the empty list should be returned by rule (i). Otherwise, by rule (ii), the result should be `[E | QS]`, with free occurrences of `v` bound to the head of the argument list, concatenated with the result of calling itself recursively with the tail of the argument list.

Thus, for the comprehension

```
[E | v <- L, QS]
```

the function

```
f$1 []     = [];
f$1 (v:l') = [E | QS] ++ f$1 l';
```

where `f$1` is a new name, should be generated, and the comprehension should then be replaced by the expression `f$1 L`.

The whole translation scheme must then be applied recursively to eliminate `[E | QS]` from the definition of `f$1`. The function that results from this should be defined in a `where`-clause within `f$1` so that the binding of free occurrences of `v` in `[E | QS]` to the head of the argument list is achieved.

The next two reduction rules show how to translate a comprehension when the first qualifier is a filter. It should be mapped to a function taking a boolean argument. If the function is applied to `False`, the result should be the empty list by rule (iii), otherwise it should be `[E | QS]` by rule (iv), where `QS` are the remaining qualifiers.

Thus for the comprehension

```
[E | B, QS]
```

where `B` is a boolean predicate, the function

```
f$2 False = [];
f$2 True  = [E | QS];
```

should be generated, and the whole comprehension replaced by `f$2 B`. Again, `f$2` is a new name, and the translation must be applied recursively to eliminate `[E | QS]`.

As an alternative, `[E | B, QS]` could be mapped to

```
if B then [E | QS] else [];
```

Which mapping that should be preferred depends on exactly how one wishes to present the evaluation of comprehensions to the user.

Finally, rule (v) shows how to end the translation process, when there are no more qualifiers, by replacing the comprehension `[E | ]` by the singleton list `[E]`.

Now, the translation scheme outlined above is far from the best one possible, but, as noted earlier, for debugging purposes, it is usually quite acceptable to sacrifice performance in order to ease debugging, and the above scheme does have the merit that it is very easy to map back the generated functions to the original source constructs.

Thus, instead of showing the user `f$1`, we should show

```
[E | v <- @, QS]
```

after having substituted whatever `f$1` is applied to for the place holder `@`. The function `f$2` should be dealt with in like manner.

In effect, the functions are given special 'debug names' in terms of the corresponding source constructs. During debugging, the user will therefore be faced by the piece of source code that the generated function represents, rather than by some obscure system generated function name, and she will be able to answer questions involving the comprehension by referring to the semantics of list comprehensions.

Note that the nested function definitions that the scheme above gives rise to, are taken care of by the normal lambda-lifting algorithm, so that bindings for free variables will show up in `where`-clauses as usual.

5.5.3 Example of List Comprehension Debugging

For a larger example, consider the following list comprehension:

```
[ x + y | x <- foo z, y <- fie z, x > y ]
```

The following function would be generated:

```
|| [ x + y | x <- @, y <- fie z, x > y]
f$0 [] = [];
f$0 (x : l) = f$1 (fie z) ++ f$0 l
              where
                 || [ x + y | y <- @, x > y ]
                 f$1 [] = [];
                 f$1 (y : l) =
                    f$2 ++ f$1 l
                    where
                       || [ x + y | x > y ]
                       f$2 = if x > y then [x + y]
                             else [];
                    ;
              ;
```

Finally `f$0 (foo z)` would be substituted for the comprehension and the function above defined in a suitable `where`-clause.

The debug names presented to the user are given as comments, whereas names involving `$` are the real, system generated names. `@` is a place holder that will be replaced by the value of the function's parameter during debugging. A typical question asked during debugging might be:

```
[ x + y | y <- [1, 3], x > y ] where x = 2
=>
[3]
Yes/No?
```

It is easily seen that the answer to this question should be yes.

A Freja program demonstrating a slightly more general list comprehension translation and generation of appropriate debug names may be found in appendix C.

5.6 Slicing

Obviously, the more information the user is able to supply the system with per question, the fewer questions the system has to ask in order to locate the bug. For example, if a user indicates that a particular element in a list is not what it is supposed to be, the system can disregard any computations that are not relevant for the production of this particular element. This technique is called *program slicing* [Wei82, Wei84] and it could reduce the number of questions asked during algorithmic debugging considerably.

For our purposes, *dynamic slicing*, in which slicing is applied to a particular execution, seems to be the most suitable. The alternative, *static slicing*, which takes account of all possible executions by statically analysing the source code, might also be a possibility, but it would be less precise since a lot of information available when given a particular execution then could not be used.

There has already been some research on using dynamic slicing in the context of algorithmic debugging for imperative languages [KSF92, Kam93], and it would thus be interesting to see how it fits into the context of algorithmic debugging for lazy functional languages. A problem here, however, is that when doing piecemeal tracing, the whole trace is not going to be available at any one time. The implications of this are as yet not clear.

5.7 Improving the User Interface

As we have seen, there are a number of fundamental technical problems that must be solved before a useful, lazy debugger can be built. While the problem of designing a good user interface might not belong to these, it should be recognized that a good user interface could be the difference between a successful debugger and a useless one. Some ideas concerning the design of such a good interface will therefore be presented in this section.

5.7.1 A Graphic User Interface

By building a window and pointer based graphic user interface, the usability of the debugger would be enhanced. Browsers could enhance the visualization of large data structures. Uninteresting parts of data structures could be folded at the click of a button, and interesting parts unfolded with equal ease. Erroneous parts of data structures could be marked, e.g. for slicing purposes.

There should also be source code browsing features, e.g. the source code for the function related to the current question could be shown in a window and source code browsers for other relevant functions brought up by clicking on function names that occurs in the question.

Some initial experiments with graphic user interfaces for algorithmic debuggers have been performed [WF93], though they were chiefly concerned with showing the execution tree graphically.

5.7.2 Hiding Details and Unnecessary Information

Frequently, the details of large data structures that are part of the arguments to functions are not important when it comes to answering questions about the correctness of the behaviour of the function. As a trivial example, consider the function to append two lists. The actual elements in the list are not important for the working of the function, something which is reflected by the polymorphic type of the function which indicates that it can append lists of any specific type.

This suggests that it might be possible to use type information and other static properties to construct heuristics regarding which parts of large data structures that are interesting and thus should be presented to the user, and which that are not and thus should be suppressed, at least initially. Considering the size of the questions with which one often is confronted when using the current system, the gain could be considerable.

As another example, consider unevaluated redexes present in the trace. Clearly they cannot have influenced the course of the execution because they have not been subject to pattern matching, nor, should they represent functions, have they been applied to any arguments. Therefore, even if these redexes are erroneous in some way, that is not the reason for the adverse behaviour of the program being debugged. Thus presentation of such redexes could be suppressed without any ill effects.

5.8 Alternative Debugging Strategies

Algorithmic debugging is a straightforward but rather inflexible debugging technique. Quite often it is also tedious, even if most redundant questions have been eliminated. (To be fair, though, debugging *per se* is not usually thought of as being a particularly intellectually stimulating and rewarding process.) A user should be allowed to use all her knowledge about possible causes of the bug symptom to quickly get to the bug. Also, it must be possible to easily try out various hypotheses or perform small experiments.

While we believe that some form of tracing is necessary for debugging purposes in the context of lazy functional languages, there are obviously a lot of different search strategies that could be implemented on top of such a trace, not only algorithmic debugging, in particular if the trace is strictified.

We also think that a piecemeal scheme is necessary to make tracing realistic. However, not having the entire trace available certainly places severe restrictions on what is possible to do. Moreover, it should be noted that the strategy for piecemeal tracing suggested earlier is biased towards algorithmic debugging, since the criteria for which parts of the trace that should be kept is based on the query distance. Other measures could be envisaged, though.

For a first example of an alternative search strategy, consider a function application that results in an execution error. Then it is usually a good idea to first have a look in the proximity of the corresponding node in the (strictified) execution tree. Thus, it would be nice to be able to quickly zoom in on that part of the tree. Even if it turns out to be ultimately futile, the loss is probably slight in comparison to the time taken to get to the area in question by algorithmic debugging, in case the bug turns out to be there. Once the suspected area of the tree has been reached, ordinary algorithmic debugging could be used.

Achieving this under a piecemeal tracing scheme would presumably not be straightforward, but it would certainly be worthwhile to try. One could imagine a two phase scheme where, during the first phase, the position of the execution error in the tree is figured out, so that tracing could be enabled in its vicinity when the program is re-executed during the second phase.

A simpler idea, would be to allow a user to search the trace for suspicious values, and jump directly to nodes in the tree where such values occur. Clearly, such a simple scheme would be superseded by dynamic slicing, if that is implemented, but it might be possible to devise a language in which more complex search criteria could be stated.

In a sense, such a facility would be orthogonal to the current assertion mechanism, which supplies answers to some questions asked in the course of algorithmic debugging, thereby suppressing them. While the assertion mechanism currently is very basic, to say the least, it could be made more flexible, perhaps by using constructs from the search language alluded to above.

Note the importance of working on a strictified trace here. Strictification ensures that all values will be seen in their most evaluated form. A particular object will therefore look the same wherever in the trace it occurs, regardless of how far it had been evaluated when it was first put into the trace. Having to phrase search criteria or assertions if this was not the case would be infeasible. Not only would it require the user to have very detailed knowledge about the evaluation process and the various stages an object passes through in the course of evaluation, but an expression covering all possible cases would also be extremely complicated and large.

In her thesis, Ducassé [Duc92] argues strongly for extensible, fully programmable debuggers, since it is more or less impossible to anticipate and cater for all possible user needs in advance. Her Prolog debugger, Opium, provides the user with a trace query language and the possibility to define abstract views of Prolog executions, thereby filtering out uninteresting, low-level execution events. The trace query language is really Prolog itself, augmented with a few primitives to get at the execution events. The abstraction facilities are also programmed in Prolog.

Now, Prolog is a strict language. Thus it is often possible to execute trace queries and perform abstraction steps on the fly, obviating the need to store any trace. This is obviously a huge advantage when it comes to debugging realistic programs. Nevertheless, in some cases it is desirable to store parts of the trace, e.g. for doing backwards tracing or providing multiple abstract views on the same piece of trace, and Opium therefore has a facility to write execution events to a trace database in chronological order.

It would be very interesting to try to adapt some of Ducassé's ideas to our lazy framework, since programmability would give the user great flexibility and e.g. search and assertion facilities would be subsumed by it.

However, one must keep the special problems of laziness, which necessitates a stored trace and strictification, in mind: unless it is possible to store the entire trace, there are limits to what is possible to achieve. Our piecemeal tracing scheme is based on properties of algorithmic debugging, and it might not be so easy to derive a similar strategy for other search strategies, particularly not if the strategy is user specified by means of a program, in which case such a derivation might have to be made automatically if the programming facilities are to be really useful.

On a slightly less exotic theme, something that would be very nice to have, and that would not be too difficult to implement, is a history mechanism, allowing the user to regret earlier answers or move back to some specific point. That would enable the user to test various hypotheses regarding the answers to difficult questions, without forcing her to redo everything if it turns out that she was wrong.

Chapter 6

Future Work

In the previous chapter, a large number of ideas for constructing a useful algorithmic debugger were presented. The fundamental ideas, i.e. thin tracing, piecemeal tracing and immediate strictification, have been developed in detail but not yet tried in practice, apart from an experimental implementation of immediate strictification.

Thus, the next step is to integrate our main ideas into a realistic prototype debugger for Freja. Apart from immediate strictification, piecemeal tracing and thin tracing, it should handle abnormally terminating as well as non-terminating programs, and it should handle some more complex source constructs such as list comprehensions. Some kind of graphic user interface would also be nice, though it is probably not worthwhile to put too much effort into this area until basic problems have been solved and it becomes clearer what kind of facilities that are needed in this respect.

Then, this debugger should be evaluated by applying it to some larger programs, which would give us valuable experience regarding the viability of the algorithmic debugging approach as such in this context, as well as regarding various implementation issues. At this stage, alternative debugging strategies, as discussed in section 5.8, could also be tried out, and cognitive aspects taken into account. Hopefully, this should lead to an improved design.

The ultimate goal, of course, is to provide a debugger for a real language, e.g. Haskell. One interesting way of doing this would be to build tracing facilities into an existing language implementation and provide a functional interface to them when compiling for debugging. The rest of the debugger (various search strategies, user interface, etc.) could then be implemented in the language in question.

For example, there could be one function that executed the program and returned an initial portion of the trace, and there could be another function that given a position in the trace, re-executed the program and produced a part of the trace rooted at that position. Or one could arrange so that parts of the trace that could not be stored are re-constructed on demand by means of the normal, lazy evaluation mechanism.

Note that this scheme is different from tracing within the language. We still have full control over what goes into the trace and how to provide a textual representation of structures on the heap. There is no problem with printing function names or forcing evaluation of unevaluated expressions.

On the theoretical side, some kind of correctness proof is needed for the tracing scheme, i.e. it should be proved that, given correct assumptions regarding which functions that are correct, and given correct answers to the questions, the erroneous function will be pin-pointed. Also, the problem of correctness assumptions and higher order functions should be investigated further.

Chapter 7

Conclusions

This thesis has argued that algorithmic debugging is a suitable technique for debugging lazy functional programs. Algorithmic debugging is a two phase technique in which first a trace in the form of an execution tree is constructed, which is then systematically searched for the bug. We believe algorithmic debugging is suitable for lazy functional languages, since these in many respects are declarative and since algorithmic debugging is itself a 'declarative' technique. Thus there is a match between the abstraction level during program development and during debugging.

An algorithmic debugger, LADT, has been implemented for a lazy functional language, and has been found to be useful in debugging some program examples. A process called *strictification* is performed on the execution tree to give the user an impression of strict evaluation. This makes the debugging process independent of the complexity of lazy evaluation order, and also helps the user to focus on the high-level declarative semantics of the application program.

The key point here, is that we believe it imperative to abstract away from low level operational issues for debugging purposes. In the case of lazy functional languages, some form of tracing seems inevitable to achieve this. This is also indicated by other work in this field, which is mostly based on tracing in some guise or another.

However, as it stands, LADT cannot be considered to be a practically usable system, mainly due to the prohibitively large trace size for any real world problem and because far too many questions have to be answered during debugging, but also technical problems with the currently employed post-execution strictification method.

Thus a number of techniques for alleviating these and other problems, as well as ideas for generally improving the debugger, were presented. The key suggestions for building a practically useful debugger are:

- *thin tracing*, so that only relevant function applications are traced,

- *piecemeal tracing*, making it possible to handle traces of arbitrary size at the expense of re-executing the program when more trace is needed, and

- *immediate strictification*, enabling thin and piecemeal tracing to be integrated with strictification and overcoming some of the deficiencies of the original post-execution strictification method.

As yet, these ideas have only been tried on a very limited scale, even though the principles have been developed in detail. It therefore remains to be seen how well the above suggestions will fare in practice.

To cut down the number of questions asked, it was argued that, apart from thin tracing, the debugger should employ *dynamic slicing*. The idea is that by being more specific about what is wrong, it should be possible to filter out many irrelevant questions.

Aspects of the user interface were also discussed, and one suggestion, among others, was that heuristics based on type information and other statically inferable properties should be used to perform sensible pretty printing of questions to reduce their size. Facilities for easy browsing of source code and large data structures are also important.

If all these techniques and ideas can be successfully integrated, we think that a practically useful algorithmic debugger for lazy functional languages could be built. We intend to do so in the near future, and then apply the debugger to some realistic problems to gain more experience.

Finally, some cognitive aspects were mentioned and alternative debugging strategies briefly considered. We think that it is quite important not to focus squarely on algorithmic debugging, but strive to give the user as much flexibility as possible. Since the construction of a strictified tree is a quite general idea, it should be possible to adapt the suggested tracing techniques to suit other debugging strategies, though exactly what is needed and how one should go about providing it is not as yet obvious. However, we hope that this will become clearer when more experience is gained form algorithmic debugging of larger programs.

Appendix A

Freja

This appendix gives the syntax and some other details on Freja [Nil91]. The syntax of Freja should be familiar to any one who knows Miranda. The most obvious differences are perhaps that Freja is not sensitive to layout, which means that definitions have to be explicitly terminated by a semicolon, and that Freja uses `==` rather than `=` for testing equality. An introduction to Miranda can be found in Peyton Jones [PJ87, pp. 431–438] or in Turner [Tur85].

A.1 Notational Conventions

Terminal symbols are typeset in typewriter font. Enclosing in curly brackets ("{" and "}"), indicates repetition of the enclosed items zero or more times, or one or more times if the closing bracket is followed by a superscript "+". Square brackets ("[" and "]") enclose optional items and the vertical bar ("|") means "or". Juxtaposition and enclosing within brackets have higher precedence than the vertical bar. Parentheses are used to change the precedences where necessary. Ambiguities are solved by taking relative precedences and associativity of operators into account.

A.2 Syntax

compilation-unit	$\rightarrow$	{ (*definition* | *declaration*) `;` }$^+$
definition	$\rightarrow$	*var-def* | *type-def*
var-def	$\rightarrow$	*fun-form* `=` *rhs* | *pattern* `=` *rhs*
fun-form	$\rightarrow$	*identifier* { *simple-ptn* }$^+$
pattern	$\rightarrow$	*simple-ptn*
	|	`-` *natural*
	|	*identifier* { `+` *natural* }$^+$
	|	*pattern* : *pattern*
	|	*IDENTIFIER* { *simple-ptn* }$^+$
simple-ptn	$\rightarrow$	*natural*
	|	*character*
	|	*string*
	|	*identifier*
	|	`(` [*ptn-list*] `)`
	|	`[` [*ptn-list*] `]`
ptn-list	$\rightarrow$	*pattern* { `,` *pattern* }
rhs	$\rightarrow$	(*expression* | *alternatives*) [*local-defs*]
alternatives	$\rightarrow$	*alternative* { `=` *alternative* } [`=` *last-alt*]
alternative	$\rightarrow$	*expression* `,` `if` *expression*
last-alt	$\rightarrow$	*expression* `,` `otherwise`
local-defs	$\rightarrow$	`where` { (*var-def* | *declaration*) `;` }$^+$

expression → { *simple-exp* }$^+$
| *unop expression*
| *expression binop expression*

unop → `-` | `~` | `#`

binop → `+` | `-` | `*` | `/` | `mod`
| `<` | `<=` | `==` | `~=` | `>=` | `>`
| `&` | `\/`

type-def → `type` *type-form* `=` *constrs*

constrs → *constr* { `|` *constr* }

constr → *IDENTIFIER* { *simple-type* }

declaration → *identifier* `::` *type*

type → *simple-type*
| *identifier* { *simple-type* }$^+$
| *type* `->` *type*

simple-type → *identifier*
| *type-var*
| `(` [*type-list*] `)`
| `[` *type* `]`

type-list → *type* { `,` *type* }

type-form → *identifier* { *type-var* }$^+$

A.3 Lexical Syntax

```
compilation-unit  →  { token | layout }+

layout            →  space | tab | nl | ff | cr | comment

comment           →  || { comment-chr } (nl | cr)

token             →  identifier | IDENTIFIER | type-var | natural
                  |  character | string | other

identifier        →  letter { idr-tail-chr }

IDENTIFIER        →  LETTER { idr-tail-chr }

type-var          →  { * }+ | ** { digit }+

natural           →  { digit }+

character         →  ' ( visible | escape ) '

string            →  " { visible | escape } "

escape            →  \ (" | ' | ? | \ | a | b | f | n | r | t | v
                  |  ascii-code

other             →  if | mod | otherwise | type | where
                  |  # | & | ( | ) | * | + | ++ | , | -
                  |  -> | / | : | :: | ; | < | <= | = | ==
                  |  > | >= | [ | \/ | ] | ~ | ~=

LETTER            →  A | B | ... | Z

letter            →  a | b | ... | z

digit             →  0 | 1 | ... | 9

idr-tail-chr      →  LETTER | letter | digit | _ | '
```

The symbols `space`, `tab`, `nl`, `ff` and `cr` stand for the literal characters space, tab, newline, form feed and carriage return. *comment-chr* stands for any character except newline and carriage return. *visible* means any visible ASCII character except single and double quotes and backslash. *ascii-code* is a natural number (decimal), consisting of one, two or three digits, in the range 0 to 255 inclusive. The escape conventions for character and string literals are as in C, except that a decimal (rather than octal) number are used to represent a character number, and that the newline character cannot be escaped.

Note that `if`, `mod`, `otherwise`, `type` and `where` are reserved words and therefore cannot be used as identifiers.

A.4 Operators and Precedences

The operators defined in Freja, their precedences and associativity are listed and briefly explained below. Precedences are given by means of a number between 1 and 9. The higher the number, the higher the precedence. Operators are either infix left associative (l), infix right associative (r), prefix unary (u), or infix relational operators allowing continued relations (e.g. `1 < a < 10`) (c). Note that type variables are written `*`, `**`, `***`, etc. in Freja.

Prec.	**Op.**	**Type**	**Explanation**
9 (u)	`#`	`[*]->num`	length of list
8 (l)	`*`	`num->num->num`	multiplication
	`/`	`num->num->num`	division
	`mod`	`num->num->num`	remainder
7 (u)	`-`	`num->num`	negation
6 (l)	`+`	`num->num->num`	addition
	`-`	`num->num->num`	subtraction
5 (c)	`<`	`*->*->bool`	less than
	`<=`	`*->*->bool`	less or equal to
	`==`	`*->*->bool`	equal to
	`~=`	`*->*->bool`	not equal to
	`>=`	`*->*->bool`	greater or equal to
	`>`	`*->*->bool`	greater than
4 (u)	`~`	`bool->bool`	boolean not
3 (l)	`&`	`bool->bool->bool`	boolean and
2 (l)	`\/`	`bool->bool->bool`	boolean or
1 (1)	`:`	`*->[*]->[*]`	list construction
	`++`	`[*]->[*]->[*]`	list concatenation

A.5 The Standard Environment

The predefined environment contains only the seven functions `hd`, `tl`, `fst`, `snd`, `shownum`, `showchar` and `showbool`; the boolean constructors `False` and `True`; and the three basic types `num`, `char` and `bool`. The predefined functions and constructors are described below. A brief description of each function is given, followed by the type declaration and formal definition of the function, expressed in Freja, where appropriate.

`hd` returns the head of a non-empty list.

```
hd::[*]->*;
hd (x:xs) = x;
```

`tl` returns the tail of a non-empty list.

```
tl::[*]->[*];
tl (x:xs) = xs;
```

`fst` returns the first component of a two-tuple.

```
fst::(*, **)->*;
fst (x, y) = x;
```

`snd` returns the second component of a two-tuple.

```
snd::(*, **)->**;
snd (x, y) = y;
```

`shownum` converts an integer to printable form, ie to a list of characters.

```
shownum::num->[char];
<defined internally>
```

`showchar` converts a character to its textual representation.

```
showchar::char->[char];
showchar c = ['\'', c, '\''];
```

`showbool` converts a boolean value to printable form.

```
showbool::bool->[char];
showbool False = "False";
showbool True = "True";
```

The constructors `False` and `True` represent the boolean values false and true.

```
type bool = False | True;
```

Appendix B

Debugging of a Simple Scanner

In this appendix a larger example is presented. The program below is a simple scanner. The bug is in the function `showtoken`; the case for `Assign` is missing.

```
|| Simple Scanner, mk II
||
|| Test program for Algorithmic Debugger
||
|| Ver. 1.00, 1993-05-02

|| ---------------------------------------------------------
|| Character recognizers
|| ---------------------------------------------------------

is_digit x = '0' <= x <= '9';

is_upper x = 'A' <= x <= 'Z';

is_lower x = 'a' <= x <= 'z';

is_alpha x = is_upper x \/ is_lower x;

is_alphanum x = is_alpha x \/ is_digit x;

is_idrtail x = is_alphanum x \/ x == '_';

|| ---------------------------------------------------------
|| Standard routines
|| ---------------------------------------------------------
```

```
dropwhile p [] = [];
dropwhile p (x:xs) = dropwhile p xs, if (p x)
                   = (x:xs), otherwise;

takewhile p [] = [];
takewhile p (x:xs) = x:(takewhile p xs), if (p x)
                   = [], otherwise;

map f [] = [];
map f (x:xs) = (f x):map f xs;

layn [] = [];
layn (x:xs) = x ++ "\n" ++ layn xs;

foldl op a [] = a;
foldl op a (x:xs) = foldl op (op a x) xs;

|| ---------------------------------------------------------
|| Conversion routines
|| ---------------------------------------------------------

char_to_num '0' = 0;
char_to_num '1' = 1;
char_to_num '2' = 2;
char_to_num '3' = 3;
char_to_num '4' = 4;
char_to_num '5' = 5;
char_to_num '6' = 6;
char_to_num '7' = 7;
char_to_num '8' = 8;
char_to_num '9' = 9;

|| ---------------------------------------------------------
|| Datatypes
|| ---------------------------------------------------------

type token = Err [char] | Int num | Idr [char] | Plus |
             Minus | Times | Divide | Leftpar | Rightpar |
             Semicol | Assign | Print | Eof;

is_error (Err msg) = True;
is_error x         = False;

is_eof x = (x == Eof);
```

```
|| -------------------------------------------------------
|| Show functions
|| -------------------------------------------------------

showtoken (Err cs) = "Error: " ++ cs;
showtoken (Int n) = "Int " ++ shownum n;
showtoken (Idr cs) = "Idr " ++ cs;
showtoken Plus = "Plus";
showtoken Minus = "Minus";
showtoken Times = "Times";
showtoken Divide = "Divide";
showtoken Leftpar = "Leftpar";
showtoken Rightpar = "Rightpar";
showtoken Semicol = "Semicol";
showtoken Print = "Print";
showtoken Eof = "Eof";

show_tokens ts = layn (map showtoken ts);

|| -------------------------------------------------------
|| Scanner
|| -------------------------------------------------------

scan_token::[char]->(token, [char]);
scan_token []          = (Eof, []);
scan_token (' ':xs)     = scan_token xs;
scan_token ('\n':xs)    = scan_token xs;
scan_token ('\t':xs)    = scan_token xs;
scan_token ('|':'|':xs) = scan_token (dropwhile neqlf xs)
                          where neqlf x = (x ~= '\n');;
scan_token ('+':xs)     = (Plus, xs);
scan_token ('-':xs)     = (Minus, xs);
scan_token ('*':xs)     = (Times, xs);
scan_token ('/':xs)     = (Divide, xs);
scan_token ('(':xs)     = (Leftpar, xs);
scan_token (')':xs)     = (Rightpar, xs);
scan_token (';':xs)     = (Semicol, xs);
scan_token (':':'=':xs) = (Assign, xs);
scan_token ('P':'R':'I':'N':'T':xs) = (Print, xs);
scan_token (x:xs)       = scan_int (x:xs), if (is_digit x)
                        = scan_idr (x:xs), if (is_alpha x)
                        = (Err ("'" ++ [x] ++
                                "' not allowed."),
                           xs), otherwise;
```

```
scan_int::[char]->(token, [char]);
scan_int xs = (Int val, rest)
        where
           val = foldl mul10add 0
                       (map char_to_num
                            (takewhile is_digit xs));
           rest = dropwhile is_digit xs;
           mul10add x y = 10 * x + y;
        ;

scan_idr::[char]->(token, [char]);
scan_idr (x:xs) = (Idr (x:idrtail), rest)
        where
           idrtail = takewhile is_idrtail xs;
           rest = dropwhile is_idrtail xs;
        ;

scan::[char]->[token];
scan xs = [tok], if is_eof tok
        = tok : (dropwhile is_error (scan rest)),
          if (is_error tok)
        = tok : (scan rest), otherwise
        where
           (tok, rest) = scan_token xs;
        ;

|| ---------------------------------------------------------
|| Main routine
|| ---------------------------------------------------------

input = "a := 2; b := 5; a := b * (a + b); PRINT a;";

main = show_tokens (scan input);
```

The interaction between the user and the debugger that follows when the above program is executed is given below. As can be seen, the program only outputs the first token and then gives a runtime error (`PE`, Program Error). The user starts by loading a few assertions and then lists them. Then he starts debugging. The commands `str` and `nostr` switch between printing lists of characters as strings and as lists of characters, while the command `uet` turns on printing of unevaluated list tails.

```
Idr a
PE
Program reduced to Program Error (ie aborted).
Fixing execution tree, will then start algorithmic debugging.
```

```
---------------------------------------
Did the program behave correctly?
(y/n/m/h)? load scanner.assert
Assertions loaded from file scanner.assert.
(y/n/m/h)? list
dropwhile
takewhile
map
layn
foldl
char_to_num
(y/n/m/h)? n

---------------------------------------
$main
=>
"Idr a\n..."
(y/n/m/h)? nostr
(y/n/m/h)? uet
(y/n/m/h)? resee
$main
=>
['I','d','r',' ','a','\n':ERROR (execution aborted)]
(y/n/m/h)? n

---------------------------------------
main
=>
['I','d','r',' ','a','\n':ERROR (execution aborted)]
(y/n/m/h)? n

---------------------------------------
input
=>
['a',' ',':','=',' ','2',';',' ','b',' ',...]
(y/n/m/h)? y

---------------------------------------
(scan ['a',' ',':','=',' ','2',';',' ','b',' ',...])
=>
[(Idr ['a']):
 (scan [' ',':','=',' ','2',';',' ','b',' ',':',...])]
(y/n/m/h)? str
(y/n/m/h)? resee
```

```
(scan "a := 2; b ...")
=>[(Idr "a"):(scan " := 2; b :...")]
(y/n/m/h)? y

----------------------------------------
(show_tokens [(Idr "a"):(scan " := 2; b :...")])
=>
"Idr a\n..."
(y/n/m/h)? nostr
(y/n/m/h)? resee
(show_tokens [(Idr ['a']):
 (scan [' ',':','=',' ','2',';',' ','b',' ',':',...])])
=>
['I','d','r',' ','a','\n':ERROR (execution aborted)]
(y/n/m/h)? n

----------------------------------------
(is_idrtail ' ')
=>
False
(y/n/m/h)? y

----------------------------------------
(showtoken (Idr ['a']))
=>
['I','d','r',' ','a']
(y/n/m/h)? y

----------------------------------------
(scan [' ',':','=',' ','2',';',' ','b',' ',':',...])
=>
[Assign:(scan [' ','2',';',' ','b',' ',':','=',' ','5',...])]
(y/n/m/h)? y

----------------------------------------
(showtoken Assign)
=>
ERROR (execution aborted)
(y/n/m/h)? n
Bug located in function showtoken.
```

Appendix C

Debugging List Comprehensions

In this appendix, a Freja program that demonstrates how list comprehensions may be compiled in a way that makes it possible to debug them algorithmically is presented. The code here is deals with slightly more general list comprehensions than were discussed in section 5.5, since patterns are allowed to the left of the arrow `<-` in generators.

```
|| Demonstration of debug-compilation of list comprehensions.

|| ----------------------------------------------------------
|| Datatypes
|| ----------------------------------------------------------

|| A comprehension is an expression and a list of generators
|| and predicates. The expression is represented by its
|| source text, which is all that is needed for this demo.

type comprehension = MkCompr [char] [gen_or_pred];

|| A generator is a pattern and an expression, both repres-
|| ented by their source text, which is all that is needed
|| for this demo. A predicate is also represented by source
|| text.

type gen_or_pred = MkGen [char] [char] | MkPred [char];
```

```
|| A function generated by the list comprehension compiler.
|| It is represented by its debugname (the function name that
|| the user will see during debugging) and a list of triples
|| where each triple represents one defining equation of the
|| function. For each equation, the source text of the LHS,
|| the source text of the RHS and any functions defined in a
|| where-clause are given.

type function = MkFun [char] [([char], [char], [function])];

|| ----------------------------------------------------------
|| Standard functions
|| ----------------------------------------------------------

map f []       = [];
map f (x : xs) = f x : map f xs;

foldl op a []     = a;
foldl op a (x:xs) = foldl op (op a x) xs;

take n       []       = [];
take 0       (x : xs) = [];
take (n + 1) (x : xs) = x : (take n xs);

repeat x = xs
           where
              xs = x : xs;
           ;

|| ----------------------------------------------------------
|| Comprehension compilation
|| ----------------------------------------------------------

|| tc translates a list comprehension. It takes an integer
|| (ns) acting as a name supply and the comprehension as its
|| parameters. It returns a pair consisting of the source
|| text to replace the comprehension by and the generated
|| comprehension functions (currently one or none) to be de-
|| fined in a where-clause within whose scope the occurrence
|| of the comprehension is situated.
```

```
tc ns (MkCompr e (MkGen gp ge : rest)) =
   ((funName ns) ++ " (" ++ ge ++ ")",
    [MkFun (debugName (MkCompr e (MkGen gp ge : rest)))
           [(funName ns ++ " [] =", "[]", []),
    (funName ns ++ " (" ++ gp ++ " : l) =",
             repl_expr ++ " ++ " ++ funName ns ++ " l",
     where_defs),
    (funName ns ++ " (_ : l) =",
             funName ns ++ " l", [])
           ]])
   where
      (repl_expr, where_defs) = tc (ns+1) (MkCompr e rest);
   ;

tc ns (MkCompr e (MkPred pe : rest)) =
   (funName ns,
    [MkFun (debugName (MkCompr e (MkPred pe : rest)))
           [(funName ns ++ " =",
    "if " ++ pe ++ " then " ++ repl_expr ++
            " else []",
    where_defs)]])
   where
      (repl_expr, where_defs) = tc (ns+1) (MkCompr e rest);
   ;

tc ns (MkCompr e []) = ("[" ++ e ++ "]", []);

|| debugName takes a list comprehension and generates a
|| printable "name" for the function that will be generated
|| when the comprehension is translated by tc. The "name" is
|| very similar to the original source code of the compre-
|| hension, enabling debugging at almost source code level.
|| A "@" in a generated name is intended to be substituted
|| for the value of the parameter to the function during
|| debugging.

debugName (MkCompr e (MkGen gp ge : rest)) =
   "[ " ++ e ++ " | " ++ gp ++ "<-" ++ "@" ++
   printGensAndPreds rest ++ " ]";

debugName (MkCompr e (MkPred pe : rest)) =
   "[ " ++ e ++ " | " ++ pe ++
   printGensAndPreds rest ++ " ]";
```

```
printGensAndPreds [] = [];
printGensAndPreds (MkGen gp ge : rest) =
   ", " ++ gp ++ "<-" ++ ge ++ printGensAndPreds rest;
printGensAndPreds (MkPred pe : rest) =
   ", " ++ pe ++ printGensAndPreds rest;

|| Generates a new "unique" function name given a name
|| supply.

funName ns = "f$" ++ shownum ns;

|| Print a generated function in source text format.

printFunction i (MkFun dn eqs) =
   indent i ++ "|| " ++ dn ++ "\n" ++ printEqns eqs
   where
      printEqns [] = [];
      printEqns ((lhs, rhs, wheredefs) : eqs) =
         indent i ++ lhs ++ " " ++ rhs ++
         printWhereDefs i' wheredefs ++
 printEqns eqs
 where
    i' = i + #lhs + 1;;
      printWhereDefs i [] = ";\n";
      printWhereDefs i whereDefs =
         "\n" ++ indent i ++ "where\n" ++
 foldl append [] (map (printFunction (i+3)) whereDefs)
         ++ indent i ++ ";\n"
         where
            append x y = x ++ y;;
   ;

indent i = take i (repeat ' ');

|| [ x + y | (x, 2) <- foo z, (y, 3) <- fie z, x > y ]

zf = MkCompr "x + y" [MkGen "(x, 2)" "foo z",
                      MkGen "(y, 3)" "fie z",
                      MkPred "x > y"];

main = repl_expr ++ "\nwhere\n" ++ printFunction 0 function
       where
          (repl_expr, [function]) = tc 0 zf;
       ;
```

The program above yields the following output when run (the output has been reformatted slightly to make it fit the page):

```
f$0 (foo z)
where
|| [ x + y | (x, 2)<-@, (y, 3)<-fie z, x > y ]
f$0 [] = [];
f$0 ((x, 2) : l) = f$1 (fie z) ++ f$0 l
                   where
                      || [ x + y | (y, 3)<-@, x > y ]
                      f$1 [] = [];
                      f$1 ((y, 3) : l) = f$2 ++ f$1 l
                                         where
                                            || [ x+y | x>y ]
                                            f$2 = if x > y
                                                  then
                                                     [x + y]
                                                  else
                                                     [];
                                         ;
                      f$1 (_ : l) = f$1 l;
                   ;
f$0 (_ : l) = f$0 l;
```

Bibliography

[Aug84] Lennart Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 218–227, August 1984.

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languages part II.* PhD thesis, Department of Computer Science, Chalmers University of Technology, December 1987.

[Bar84] Henk P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 2nd edition, 1984.

[BMS80] Rod Burstall, David MacQueen, and Don Sanella. HOPE: An experimental applicative language. Report CSR-62-80, Computer Science Department, Edinburgh University, 1980.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[Car84] Luca Cardelli. Compiling a functional language. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 208–217. ACM Press, August 1984.

[CH93] Magnus Carlsson and Thomas Hallgren. Fudgets: A graphical user interface in a lazy functional language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 321–330, Copenhagen, Denmark, June 1993. ACM Press.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Duc92] Mireille Ducassé. *An Extendable Trace Analyser to Support Automated Debugging.* PhD thesis, University of Rennes I, Campus de Beaulieu, 35042 Rennes cedex, France, June 1992. Numéro d'ordre 758. European Doctorate. In English.

[FGKS92] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmehri. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, 1(4):303–322, December 1992.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley Publishing Company, 1988.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[HK84] Paul Hudak and David Kranz. A combinator-based compiler for a functional language. In *Proceedings of the 11th ACM Annual Symposium on Principles of Programming Languages*, pages 122–132. ACM Press, January 1984.

[HKS91] Wolfgang Henhapl, Stefan Kaes, and Gregor Snelting. Utilizing fifth generation technology in software development tools. Report PI-R3/91, Technische Hochschule Darmstadt, March 1991.

[HM93] Jonathan E. Hazan and Richard G. Morgan. The location of errors in functional programs. In Peter Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 135–152, Linköping, Sweden, May 1993.

[HO85] Cordelia V. Hall and John T. O'Donnell. Debugging in a side effect free programming environment. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 60–68, Seattle, Washington, June 1985. Proceedings published in ACM SIGPLAN Notices 20(7).

[HPJW+92] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Huges, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[Hud89] Paul Hudak. Conception, evaluation and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Hug89] John Hughes. Why fuctional programming matters. *The Computer Journal*, 32(2):98–197, April 1989.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the 1984 ACM SIGPLAN Symposium on*

Compiler Construction, pages 58–69, June 1984. Proceedings published in ACM SIGPLAN Notices, 19(6).

[Joh87] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Science, Chalmers University of Technology, February 1987.

[Kam90] Samuel Kamin. A debugging environment for functional programming in Centaur. Research report, Institut National de Recherche en Informatique et en Automatique (INRIA), Domaine de Voluceau, Rocquencourt, B.P.105, 78153 Le Chesnay Cedex, France, July 1990.

[Kam93] Mariam Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing.* PhD thesis, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, April 1993.

[KHC91] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991.

[Kis92] Amir Kishon. *Theory and Art of Semantics Directed Program Execution Monitoring.* PhD thesis, Yale University, May 1992.

[KSF92] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384, Leuven, Belgium, August 1992.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[LPJ94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994. ACM Press. To appear.

[LSB94] Bengt Lennartsson, Nahid Shahmehri, and Staffan Bonnier. On exploiting declarative programming and parallel execution in computer based real-time systems. In *Proceedings of the IEEE Workshop on Systems Engineering of Computer based Systems, ECBS '94.* IEEE, May 1994.

[Mil84] Robin Milner. A proposal for standard ML. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984. ACM Press.

[Nai92] Lee Naish. Declarative debugging of lazy functional programs. Research Report 92/6, Department of Computer Science, University of Melbourne, Australia, 1992.

[NF92] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 385–399, Leuven, Belgium, August 1992.

[NF93] Henrik Nilsson and Peter Fritzson. Lazy algorithmic debugging: Ideas for practical implementation. In Peter Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 117–134, Linköping, Sweden, May 1993.

[NF94] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3), July 1994. To appear.

[Nil91] Henrik Nilsson. Freja: A small non-strict, purely functional language. MSc dissertation, Department of Computer Science and Applied Mathematics, Aston University, Birmingham, UK, September 1991.

[OH88] John T. O'Donnell and Cordelia V. Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Sha82] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, May 1982.

[Sha91] Nahid Shahmehri. *Generalized Algorithmic Debugging*. PhD thesis, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, 1991.

[Spa94] Jan Sparud. An embryo to a debugger for Haskell. Presented at the annual internal workshop "Wintermötet", held by the Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, January 1994.

[Sto77] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Stu92] Robert Sturrock. Debugging systems for lazy functional programming languages. Honours dissertation, Department of Computer Science, University of Melbourne, Australia, November 1992.

[Szy91] Boleslaw K. Szymanski, editor. *Parallel Functional Languages and Compilers.* ACM Press, 1991.

[Toy87] Ian Toyn. *Exploratory Environments for Functional Programming.* PhD thesis, Department of Computer Science, University of York, York, UK, April 1987.

[TR86] Ian Toyn and Collin Runciman. Adapting combinator and SECD machines to display snapshots of functional computations. *New Generation Computing*, 4(4):339–363, 1986.

[Tur85] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture, FPCA'85*, volume 201 of *Lecture Notes in Computer Science*, Nancy, 1985.

[Wad71] Christopher Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* Phd thesis, Oxford, 1971.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[WF93] Rickard Westman and Peter Fritzson. Graphical user interfaces for algorithmic debugging. In Peter Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 273–286, Linköping, Sweden, May 1993.